RealGraph: A Graph Engine Leveraging the Power-Law Distribution of Real-World Graphs

Yong-Yeon Jo
Hanyang University
Seoul, Korea
jyy0430@hanyang.ac.kr

Myung-Hwan Jang
Hanyang University
Seoul, Korea
sugichiin@hanyang.ac.kr

Sang-Wook Kim*
Hanyang University
Seoul, Korea
wook@hanyang.ac.kr

Sun-Ju Park
Yonsei University
Seoul, Korea
boxenju@yonsei.ac.kr

ABSTRACT

As the size of real-world graphs has drastically increased in recent years, a wide variety of graph engines have been developed to deal with such big graphs efficiently. However, the majority of graph engines have been designed without considering the power-law degree distribution of real-world graphs seriously. Two problems have been observed when existing graph engines process real-world graphs: inefficient scanning of the sparse indicator and the delay in iteration progress due to uneven workload distribution. In this paper, we propose RealGraph, a single-machine based graph engine equipped with the hierarchical indicator and the block-based workload allocation. Experimental results on real-world datasets show that RealGraph significantly outperforms existing graph engines in terms of both speed and scalability.

CCS CONCEPTS

• **Information systems** → *Graph-based database models*; *Database management system engines*; *Computing platforms*.

KEYWORDS

Graph engine, single machine, real-world graph, power-law degree distribution

ACM Reference Format:
Yong-Yeon Jo, Myung-Hwan Jang, Sang-Wook Kim, and Sun-Ju Park. 2019. RealGraph: A Graph Engine Leveraging the Power-Law Distribution of Real-World Graphs. In *Proceedings of the 2019 World Wide Web Conference (WWW '19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3308558.3313434

1 INTRODUCTION

Graphs are widely used data structures for representing relationships between objects in various real-world domains, such as the web, social networks, protein networks, etc. Recently, with their increasing usage, the size of real-world graphs has increased rapidly. Such real-world graphs are called *big graphs*. To meet the demand for efficient handling of real-world big graphs, various graph engines have been developed [1, 6, 18, 25, 30]. These graph engines provide a simple programming interface to help programmers easily develop and execute graph algorithms for big graphs [14, 21].

Since all graph algorithms could be iteratively performed, existing graph engines adopt an *iterative processing* method, where an iteration starts to proceed *after* all the nodes to be processed at its previous iteration are completely processed [8, 19, 24]. Each iteration consists of two phases. At the first phase, the graph engine scans the *indicator* consisting of a bit vector [14, 29] to identify the nodes to be processed. At the second phase, it processes the identified nodes in parallel through multi-threading. Both phases of the iteration in existing graph engines, however, have been developed without careful considerations on the degree distribution of real-world graphs.

Most real-world graphs follow the *power-law degree distribution*, which is characterized by very few nodes having a very high degree (i.e., *hub nodes*), while most nodes (i.e., non-hub nodes) having a low degree [15]. The power-law degree distribution of real-world graphs causes following two phenomena when existing graph engines execute graph algorithms. First, the distribution of the nodes processed at iterations is skewed where the majority of nodes are processed at less than first half iterations while only a small number of nodes are processed at the remaining iterations[1]. Second, at some iterations, the nodes with very large degree disparity (i.e., hub and non-hub nodes) are processed together.

These phenomena cause the following two problems, which slows down the performance of the graph engines. The problem related to the first phase of the iteration is *inefficient scanning of the indicator*. Existing graph engines use a single flat indicator [14, 25, 29, 30]. They scan the whole indicator *linearly* at each iteration to identify the nodes to be processed. In the majority of iterations, the number of nodes to be processed tends to be very small, which makes the indicator very sparse in those iterations. Linear scanning of the very sparse indicator causes unnecessary computations and leads to the performance degradation of the graph engines.

The problem related to the second phase of the iteration is the delay for the progress of the next iteration due to *uneven workload*

*Corresponding author.

WWW '19, May 13–17, 2019, San Francisco, CA, USA

ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3313434

[1]Note that this phenomenon does not happen in the graph algorithms that processes *all nodes* at each iteration, such as PageRank.

allocation over threads. Existing graph engines adopt the node-based workload allocation, where each thread processes an individual node with its edges [6, 14, 18, 30]. Since the workload of each thread is proportional to the number of edges, the node-based allocation produces huge difference in the workload between the threads that process hub nodes and the others that process non-hub nodes. This causes the threads that process non-hub nodes to wait for the ones that process hub nodes to complete, which delays the start of the next iteration.

To solve these two problems, we propose a new single-machine based graph engine[2] called *RealGraph*. RealGraph is designed based on the principles of well-known database systems [3, 4, 7] to manage big data efficiently. Furthermore, When storing graph data, RealGraph preserves *data locality* by adopting *efficient data layout* [12] that stores the data accessed together in adjacent storage space. In addition, we propose two new techniques considering the characteristics of real-world graphs to execute the graph algorithms efficiently.

First, we propose the *hierarchical indicator* that consists of multi-level bit vectors designed to skip easily the ranges of the indicator to be unnecessarily scanned. The lowest-level bit vector is the same as the flat indicator in existing graph engines. A higher-level bit vector compresses some range of its lower-level bit vector using a single bit. A bit set as 1 of each level bit vector denotes that a corresponding range at its lower-level bit vector should be scanned. The process starts from the highest-level bit vector; it scans the range of the lower-level bit vector corresponding to the bit (if each higher-level bit vector has a bit set as 1). Unlike the existing flat indicator, the hierarchical indicator reduces unnecessary scanning of the sparse indicator dramatically.

Second, we propose the block-based workload allocation which assigns *a block* to each thread, instead of a node and its edges in the node-based one. A block is a fixed-size standard I/O unit of the graph engine, which stores multiple nodes and their edges together. Since the block size is fixed, the total number of edges stored in each block is almost the same. Thus, the graph engine can allocate almost the same workload to each thread. This reduces the waiting time among threads at every iteration, which eventually accelerates the start of the next iteration.

We show the effectiveness of the proposed techniques through extensive experiments. The results indicate that RealGraph improves the performance considerably when using the proposed techniques together rather than individually. We also demonstrate the performance of RealGraph by comparing it with five well-known single-machine based graph engines [14, 21, 25, 29, 30] and four distributed-system based graph engines [1, 9, 16, 26]. For experiments, we used seven graph algorithms and six real-world graphs. For all graph algorithms and datasets, RealGraph shows the best performance in terms of processing speed and scalability compared with existing graph engines.

The contributions of this paper are as follows.

- First, we point out the problems that occur when existing graph engines deal with real-world graphs.
- Second, we propose two techniques for efficient processing of real-world graphs: (1) the hierarchical indicator to reduce unnecessary scanning and (2) the block-based workload allocation to balance the workload among threads.
- Third, we propose RealGraph, a single-machine based graph engine employing with these techniques.
- Finally, we verify the superiority of RealGraph through extensive experiments.

The organization of the paper is as follows. Section 2 addresses our motivation. Section 3 overviews our RealGraph. Section 4 explains the hierarchical indicator. Section 5 describes the block-based workload allocation. Section 6 shows experimental results. Finally, Section 7 summarizes and concludes the paper.

2 MOTIVATION

This section describes the execution procedure of a graph algorithm in existing graph engines, explains the characteristics of real-world graphs, and points out the problems when existing graph engines process real-world graphs.

The graph engine is a general-purpose software framework that aims to process various graph algorithms, rather than targeting specific graph algorithms [6, 9, 13, 14, 16–19, 21, 25, 26, 29, 30]. Although some graph algorithms could be processed asynchronously without synchronization between threads, all graph algorithms can be executed synchronously where threads are synchronized at regular intervals [8]. Therefore, most graph engines adopt *iterative processing* based on synchronous execution[3], where all the nodes to be processed at each iteration are processed completely and then the next iteration starts [24].

At each iteration, the graph engine executes a graph algorithm with two phases [13, 14, 25]: (1) identifying the nodes to be processed and (2) processing the identified nodes and their edges. In the first phase, the graph engine identifies the nodes to be processed by scanning the *indicator* consisting of a bit vector[4] of the length equal to the number of nodes [14, 25, 30]. If the bit is set as 1, it means that the node corresponding to the bit is to be processed at the current iteration. In the second phase, threads process the identified nodes and their edges in parallel. The existing graph engines adopt the node-based workload allocation that assigns a node and its edges to each thread [6, 18, 30].

However, both phases were designed without careful consideration of the degree distribution of real-world big graphs. Note that most real-world graphs show the power-law degree distribution, consisting of a small number of hub nodes and a very large number of non-hub nodes [15].

[2]Existing graph engines can be classified into two categories: single-machine based ones and distributed-system based ones.

[3]Some graph engines adopt an asynchronous execution based processing [10, 24].

[4]For the indicator composed of a bit vector, the space required for representing a node is a single bit. Therefore, even when dealing with a big graph with a large number of nodes, the graph engine requires only a small, fixed space. The downside of using a bit vector is that many bits are left unused when only a small number of nodes are processed in an iteration. A *linked list* may be considered as an alternative data structure. The linked list, however, requires much more space to represent a single node (e.g., 4 bytes in the case of using an integer for a node). In the iteration that processes only a small number of nodes, there is an advantage of using only a very small space for the nodes. But there would be a significant overhead of allocating and releasing a number of nodes for a linked list at each iteration. In particular, if the number of nodes that need to be processed in an iteration exceeds the given memory space, the graph engine could not run. This is the reason why most graph engines employ the simple and small bit vector for the indicator.

For example, Figure 1 shows the degree distribution of the Yahoo dataset used in our experiments, where the x-axis represents the degree of a node, and the y-axis does the number of nodes having the degree. We observe this dataset also follows the power-law degree distribution. This power-law degree distribution of the real-world graph causes the following phenomenon at each phase, which degrades the performance of graph engines.

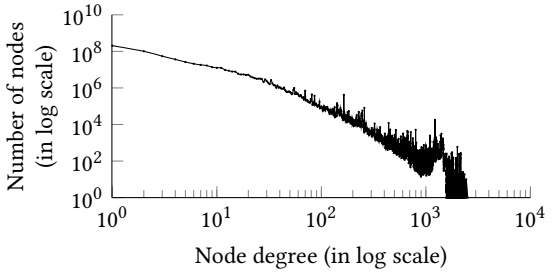

Figure 1: Degree distribution of the Yahoo dataset.

First, the distribution of the nodes processed at iterations is skewed. Since hub nodes are connected to many other nodes, they are highly likely to be accessed at early iterations in general regardless of the starting node. Also, a large number of nodes connected to the hub nodes are accessed together in the following iteration. As a result, the majority of nodes tend to be processed at early iterations, and only a small number of nodes that are multiple hop away from the hub nodes are likely to be handled at remaining iterations.

Figure 2 shows the cumulative number of the nodes processed up to each iteration, when the breadth first search (BFS) is performed on all the datasets used in experiments. Since the total number of iterations is different for each dataset, the x-axis represents the ratio of the number of processed iterations to that of the entire iterations. The y-axis indicates the cumulative percentage of the processed nodes. At early 30% or less of iterations, almost 90% of nodes are processed, and fewer than 10% nodes are processed over the remaining 70% of iterations. It is also observed that only a few nodes are processed across at more than half of the remaining iterations.

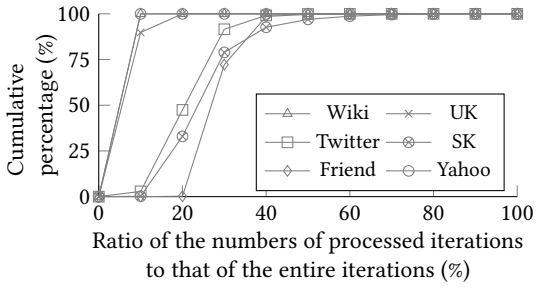

Figure 2: Cumulative number of the nodes processed up to each iteration when BFS is performed.

This phenomenon makes the indicator *very sparse in most iterations*. Since the existing graph engines linearly scan the entire indicator, the scanning becomes very inefficient. As the graph size increases, this linear scanning becomes even more impractical. In fact, with 1.4 billion nodes in the Yahoo dataset, the largest dataset used in our experiments, the scanning overhead for the indicator takes up to 50% of the total BFS execution time.

Secondly, the degrees of the nodes processed at each iteration vary significantly. In particular, the degree difference between nodes is very large at the iterations where hub nodes and non-hub nodes are processed together. Figure 3 shows the degree difference between the nodes processed at each iteration, when BFS is performed on all the datasets. The results for the first 100 iterations are shown[5]. The x-axis represents the iteration number, and the y-axis shows the ratio of the minimum node degree to the maximum node degree in log scale. We observe the existence of significant degree difference at every iteration, and in extreme cases the difference is up to 10^6 times more.

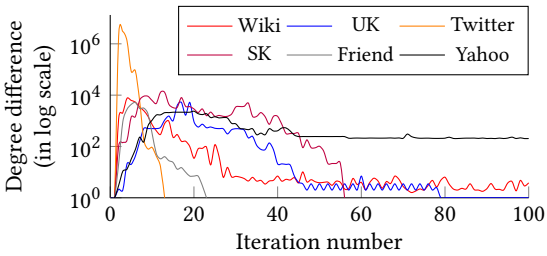

Figure 3: Degree difference between the nodes processed at each iteration when BFS is performed.

The degree difference between nodes causes uneven distribution of workload among threads in existing node-based workload allocation. Because the workload of threads is proportional to the degree of the assigned node in the node-based allocation [14, 25, 29] the workload among threads varies, which in turn delays the completion of the iteration. This is particularly true at the iteration which processes the hub nodes, because the threads with non-hub nodes should wait until the ones with the hub nodes to complete their executions. This delays the start of the next iteration, which eventually slows down the graph engine.

These observations motivate us to develop our RealGraph, a graph engine equipped with our two novel techniques: (1) efficient scanning of the sparse indicator and (2) uniform workload allocation over threads to minimize the delay for the progress of the next iteration.

3 OVERVIEW OF REALGRAPH

In this section, we describe the general architecture of our RealGraph and the efficient data layout for graph storage in RealGraph. In the following Sections 4 and 5, we describe how RealGraph solves the problems mentioned in Section 2.

3.1 Architecture

RealGraph employs, as a programming model, the *vertex-centric programming model* [14, 30] that performs a graph algorithm per the node with its edges. As a memory model, RealGraph employs the *external-memory model* [21, 29] that utilizes external storage when graph data exceeds the main memory.

[5]We show the results for the first 100 iterations, since the numbers of total iterations are different across datasets.

As shown in Figure 4, RealGraph consists of four layers: storage management layer, buffer management layer, object management layer, and thread management layer. The lower three layers manage memory and storage space. They are designed on the basis of well-known database storage systems, such as WiSS [7], Exodus [4], and Shore [3]. The thread management layer executes graph algorithms and is implemented with two techniques proposed in this paper. On top of them, we also provide a web-based user interface[6]. Using the user interface, anyone could execute a graph algorithm provided by RealGraph on their own graphs and download its result. The roles and components of each layer except for the interface are as follows.

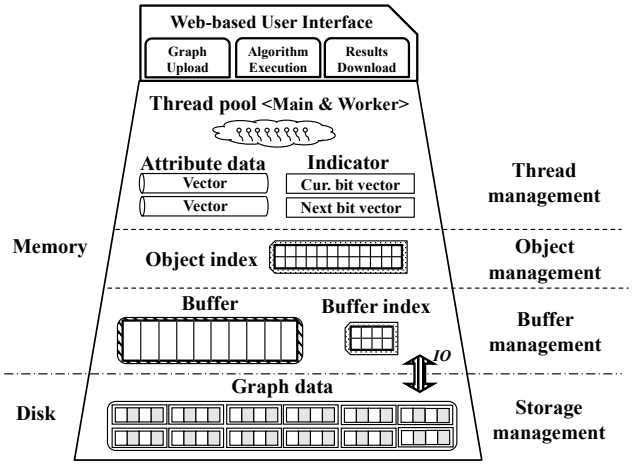

Figure 4: Architecture of RealGraph.

The storage management layer manages storage space. The graph data is divided into blocks and stored in storage space. A block is the standard I/O unit, and each block contains one or more objects. Here, an object is composed of a node and its edges[7]. If the size of an object exceeds the block size, it is stored across several blocks. This layer also manages I/Os of these blocks.

The buffer management layer manages the space for blocks in the main memory. It consists of a buffer and a buffer index. A buffer is a space with a given size in main memory and keeps blocks within that space. The buffer index has the indices for blocks loaded in buffer, which helps to access the blocks in the buffer quickly. This index is always loaded in the main memory.

The object management layer manages the location information for blocks and objects of the graph data. There is an object index which contains the indices of objects and blocks stored in storage device. It is always loaded in the main memory.

The thread management layer performs a graph algorithm. It consists of three components: thread pool, attribute data, and two indicators (i.e., current and next). The thread pool has two types of threads: worker threads that actually perform graph algorithms and one main thread that manages worker threads. The attribute data stores intermediate and final results for either nodes or edges obtained when performing a graph algorithm (e.g., nodes visited in BFS, node ranks in PageRank). The attribute data consists of several attribute vectors, each of which has the same number of elements as that of nodes. The element of each vector is the attribute corresponding to the node. If the size of the attribute data exceeds that of a given memory, they are divided into several chunks and stored in the secondary storage; each chunk required by a graph algorithm is loaded into memory. The current/next indicator stores the information about the nodes to be processed at the current/next iteration. Both indicators are always loaded in the main memory.

On the basis of the above architecture, each iteration of a graph algorithm is executed as follows. First, the main thread identifies the nodes to be processed at the current iteration by scanning the *current* indicator. Next, using the object index, the main thread finds the locations of the blocks including the objects corresponding to the identified nodes, and checks if the block exists in the buffer. Then, the main thread assigns each block to be processed to each worker thread in order. If it does not, the worker thread requests the block to the storage management layer and loads it into the buffer. After loading, the worker thread executes the function of a graph algorithm and updates the results in the corresponding attribute data. Then, it sets the next indicator such that the nodes connected to the processed nodes in the current iteration are to be processed at the next iteration. Finally, before the start of the next iteration, the next indicator is switched to the current indicator, and the current indicator is initialized to be used as the next indicator.

3.2 Efficient data layout

A good placement of nodes and edges in storage (i.e., data layout) would improve both CPU and I/O performances of a graph engine. Existing graph engines, however, have aimed at maximizing the utilization of computing resources through system optimization, overlooking the importance of data layout [14, 25].

Recognizing this, we have borrowed the concept of efficient data layout [12] and applied it to RealGraph. The main idea for this data layout is, the data accessed together in executing graph algorithms are placed in the same or adjacent storage space. Since RealGraph stores objects (each having a node and its edges) in blocks in the ascending order of node IDs as other graph engines [21, 29], we assign consecutive node IDs to the nodes likely to be accessed together at the same or successive iteration(s) by a graph algorithm. This idea makes the objects accessed together in a graph algorithm to be stored in the same or adjacent block(s).

Based on reference [12] to assign consecutive node IDs to nodes to be accessed together in various graph algorithms, breadth first search (BFS) is used as follows: BFS is performed from an arbitrary starting node in a graph given; the nodes accessed by BFS are recorded one by one in the order; the node ID used in a graph engine is determined by this order. The data layout obtained by this idea makes the nodes and edges accessed together at the same or successive iteration(s) of graph algorithms placed in the same or adjacent block(s) successfully.

This data layout transformation is performed at the preprocessing step where RealGraph stores a graph[8]. The data layout transformation is required only once for each graph to be stored in

[6] http://166.104.110.75:20080

[7] Since RealGraph uses the adjacency list, each object is stored as a node and the edges pointing to its adjacent nodes.

[8] Most graph engines also perform the preprocessing step when storing a graph. For example, GraphChi divides a graph into shards and then stores them in its preprocessing step.

RealGraph, and the graph stored in this new data layout is beneficial to the performance of many graph algorithms running on RealGraph. This implies that the transformation overhead is shared by many graph algorithms running on RealGraph.

Reference [12] reports that the data layout determined by BFS enables a graph engine not only to reduce the number of block accesses but also to face more sequential accesses of blocks, rather than random accesses, thereby improving its performance considerably.

Figure 5 shows two data layouts. A data layout has nine blocks represented by rectangles. A circle represents a node and its edges in a block. The numbers in circles indicate the order of access requests by a given graph algorithm. For processing, a graph engine should load 8 nodes into the main memory in Figure 5. In this case, by changing into the efficient data layout from existing data layout, we improve the I/O and CPU performances of RealGraph.

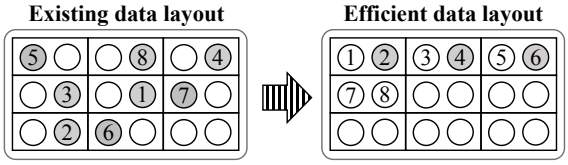

Figure 5: Data layout transformation.

When performing graph algorithms, worker threads need frequent access to blocks and attribute data. Even when it is not possible to load all of the blocks into memory, since the objects to be processed together are more likely to be located in the same or successive blocks, the number of I/O requests would be decreased and the sequential access rate could be increased. In Figure 5, RealGraph accesses only four blocks sequentially, instead of 8 blocks randomly. Similarly, the I/O performance would be increased when entire attribute data could not be loaded into memory, since the same or successive chunks of attribute data are more likely to be updated. In both cases, it is more likely to re-access the data already loaded in the main memory, which results in the increased cache hit ratio and improves the CPU performance. In addition, since adjacent bits are more likely to be set at each iteration in the current indicator, the cache hit ratio for the indicator would be increased.

Through experiments, we evaluate the performance of the efficient data layout when applied to RealGraph and also examine the synergy effects when the efficient data layout is used in combination with our two techniques in the following sections.

4 HIERARCHICAL INDICATOR

As explained in Section 2, linear scanning of the sparse indicator is very inefficient. For efficient scanning, we propose a hierarchical indicator that skips efficiently the ranges that do not need to be scanned.

The hierarchical indicator is made up of two or more bit vectors, where the upper-level bit vector determines the ranges to be scanned in the lower-level bit vector. Each level bit vector is divided into a set of ranges with a fixed length. The lowest-level bit vector is the same as the flat indicator used in existing graph engines. Each higher-level bit vector consists of a number of bits as many as the number of ranges in each lower-level bit vector. Each bit of the higher-level bit vector refers to the corresponding range of its lower-level bit vector. Both the current and next indicators are configured in the same way.

Figure 6 shows the hierarchical indicator consisting of three level bit vectors with a range length of 3. The top-level bit vector consists of three bits. The middle-level bit vector has three ranges with 9 bits. The bottom-level bit vector has a total of 9 ranges with 27 bits.

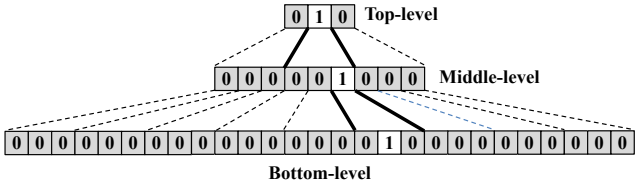

Figure 6: 3-level hierarchical indicator.

The current and next hierarchical indicators are used as follows. At each iteration, the main thread identifies the nodes to be processed in the *top-down manner* from the current hierarchical indicator. It scans each higher-level bit vector to identify the bits set as 1, and determines the ranges to be scanned of the lower-level bit vector. Then, it scans those ranges only. The main thread repeats this process and finds the nodes to be processed corresponding to the bits set as 1 in the lowest-level bit vector. By using the current hierarchical indicator, the main thread in RealGraph could skip all the ranges of the lower-level bit vector corresponding to the bits with 0 of the higher-level bit vector.

In Figure 6, the main thread scans the top-level bit vector and identifies the second bit set as 1. Only the second range of the middle-level bit vector is scanned, and the sixth bit with 1 is found. The sixth range of the bottom-level bit vector is scanned, and node 17 is identified as the node to be processed at the current iteration. In this example, the main thread scans only 9 bits, instead of all 27 bits of the bottom-level bit vector.

For the next hierarchical indicator, the worker threads set the nodes to be processed at the next iteration in the *bottom-up manner*. Each worker thread sets the bits corresponding to the nodes to be processed in the lowest-level bit vector as 1. Then, it sets the bit of the higher-level bit vector corresponding to the range including the bit set as 1 of the lower-level bit vector. This process is repeated until the top-level bit vector is reached.

Suppose Figure 6 is the next hierarchical indicator where the worker thread sets the 17th bit in the bottom-level bit vector. It also sets the sixth bit in the middle-level bit vector corresponding to the sixth range including the 17th bit in the bottom-level bit vector. Finally, it sets the second bit of the top-level bit vector corresponding to the second range containing the sixth bit in the middle-level bit vector.

To construct the hierarchical indicator, two parameters are required: the range length and the height of the hierarchical indicator. Since the height is automatically determined once the range length is fixed, the hierarchical indicator could be configured by determining the range length.

Figure 7 shows the result of BFS by changing the range length on the Yahoo dataset. Figure 7-(a) shows the scanning time, and Figure 7-(b) shows the memory usage and the height of the hierarchical indicator in levels. The experimental results show that as

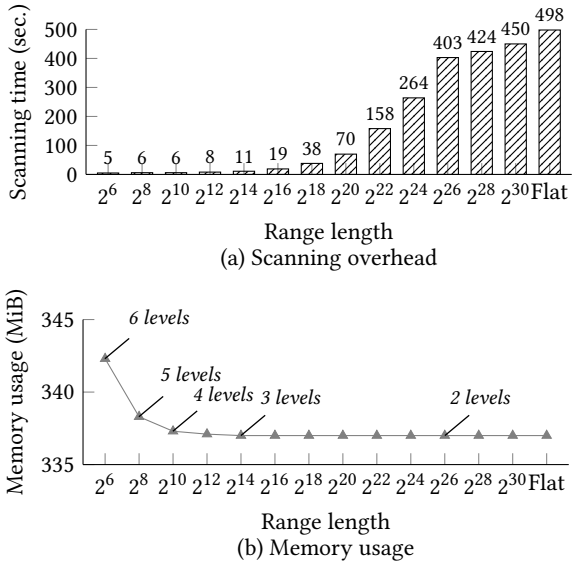

Figure 7: Scanning overhead and memory usage of the hierarchical indicator.

the range length increases, the scanning time increases, and the memory usage decreases. When considering both scanning time and memory usage, the range length of 2^{10} bits is the most efficient. The hierarchical indicator with the range length of 2^{10} bits uses only 0.1% more memory than the flat indicator employed in existing graph engines and reduces the scanning time dramatically to one-hundredth. In subsequent experiments, therefore, we use 2^{10} bits for the range length of the hierarchical indicator.

5 BLOCK-BASED WORKLOAD ALLOCATION

As mentioned in Section 2, the degree difference at each iteration causes uneven workload distribution among threads in the node-based workload allocation. This delays the progress of an iteration, which leads to degradation of the overall performance of the graph engine.

To solve the delay for the progress of the next iteration, we propose the block-based workload distribution which assigns a *block* to each thread, rather than an object (i.e., a node with its edges). A block may contain multiple objects of different sizes, but the sum of their sizes is almost the same. In RealGraph, threads are given with almost equal workloads due to the block-based workload distribution, which reduces the disparity of the completion times between threads.

Since the block size affects the performance of RealGraph, we examine the appropriate block size for RealGraph.

Figure 8 shows the execution time of PageRank[9] with 20 iterations, starting at 64KiB and increasing the block size by 4 times on the Yahoo dataset. The x-axis represents the block size, and the y-axis represents the execution time. The result confirms that the 1024KiB block shows the best performance. Thus, we use 1024KiB as the block size for RealGraph in the following experiments.

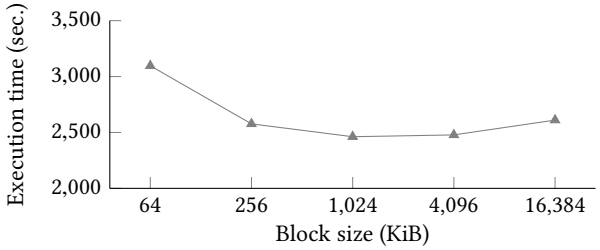

Figure 8: Execution time of PageRank with different block sizes.

To verify the superiority of the block-based workload allocation compared to the node-based one, we apply both node and block allocation techniques to RealGraph and measure the workload of each thread (i.e., the number of processed edges) at each iteration while performing PageRank on the Yahoo dataset. Figure 9 shows the result where the x-axis represents the iteration number, and the y-axis represents the standard deviation of the workload among threads. There exist a few cases where the standard deviation of the workload in the block-based allocation is larger than that in the node-based allocation. In overall, however, the block-based allocation has a much smaller standard deviation than the node-based one. This indicates the block-based allocation distributes the workload more evenly across threads.

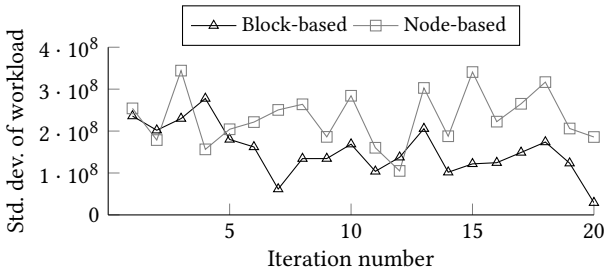

Figure 9: Standard deviation of workload over threads at each iteration

Note that the block-based workload allocation requires additional *thread synchronization* when handling a large object. A single object stored across *multiple blocks*, such as the object for a hub node, is handled by *multiple threads* in RealGraph. If multiple threads that handle a single object need to update the result of the same attribute, thread synchronization is required, making the threads progress in sequence. Since thread synchronization is an operation with large overhead, frequent thread synchronization could degrade the performance of the graph engine. If many objects are stored over several blocks, the performance of RealGraph could slow down.

Fortunately, in real-world graphs, these objects occupy only a small fraction. For all the datasets used in the experiment, when the block size is 1024KiB, only 0.0003% of the objects are stored in several blocks on average. Because the frequency of thread synchronization is very rare during the execution of the graph algorithm,

[9]BFS accesses *part of* nodes at each iteration, which results in random accesses to blocks. The random access of blocks influences some other factors such as buffer replacement policy to affect the performance of the graph algorithm. PageRank, on the other hand, causes sequential accesses to blocks, because it sequentially accesses all the nodes at each iteration. With the sequential access of blocks, other factors are less likely to affect the performance of the graph algorithms. Therefore, in this experiment, we use PageRank to observe the change of performance according to the block size as much as possible.

the performance degradation of RealGraph due to the thread synchronization is negligible.

6 PERFORMANCE EVALUATION

This section demonstrates the superiority of RealGraph through experiments. Section 6.2 validates the techniques applied to RealGraph. Section 6.3 compares the performance of RealGraph with existing single-machine based graph engines. Section 6.4 compares the performance of RealGraph with those of existing distributed-system based graph engines.

6.1 Experimental Setup

We employed five state-of-the-art single-machine based graph engines: GraphChi [14], TurboGraph [25], GridGraph [30], FlashGraph [29], and X-Stream [21]. These engines and RealGraph were performed on PCs with Intel i7-7700K, 1TiB SSD, and 64GiB memory. We also compared RealGraph with four distributed-system based graph engines: PowerGraph [9], GraphLab [16], GraphX [26], and Giraph [1].

As experimental datasets, we used six real-world graphs of varying sizes. Table 1 shows the specifications of each dataset, where *#Nodes* is the number of nodes, *#Edges* is the number of edges, and *Size* is the data size. The descriptions of datasets are as follows[10]. *Wiki* is the hyperlinked data of wiki pages in English Wikipedia. *UK* is the dataset representing the relationships of web pages having the .uk domain. *Twitter* is the dataset with the relationships of tweets collected on Twitter. *SK* is the dataset with the relationships of web pages having the .sk domain. *Friend* is the dataset with all links among users in the online gaming social network, Friendster. *Yahoo* is the hyperlinked data of web pages in Yahoo! AltaVista.

Table 1: Graph datasets

Datasets	Real-world graphs					
	Wiki	UK	Twitter	SK	Friend	Yahoo
#Nodes (M)	12	39	61	50	68	1,413
#Edges (B)	0.37	0.93	1.4	1.9	2.5	6.6
Size (GiB)	5.7	16	24	32	44	114

We employed seven graph algorithms widely used in various fields: breadth first search (BFS) [22], weakly connected component (WCC) [23], betweenness centrality (BC) [2], PageRank [20], random walk with restart (RWR) [28], belief propagation (BP) [11], and sparse matrix and vector multiplication (SpMV) [27]. Since BFS, WCC, and BC are the algorithm that accesses *part of* nodes at each iteration, the indicator is used. For the remaining four graph algorithms (i.e., PageRank, RWR, BP, and SpMV), the indicator is not used, because they access all the nodes at each iteration [11]. Since BFS, WCC, and BC differ in the number of iterations according to the datasets, they are performed until their termination conditions are satisfied. For the other four graph algorithms (i.e., PageRank, RWR, BP, and SpMV), the number of iterations is fixed as 20.

[10] Wiki: http://konect.uni-koblenz.de/networks/wikipedia_link_en, UK: http://law.di.unimi.it/webdata/uk-2005, Twitter: an.kaist.ac.kr/traces/WWW2010, SK: http://law.di.unimi.it/webdata/sk-2005, Friend: https://archive.org/details/friendster-dataset-201107, Yahoo: http://webscope.sandbox.yahoo.com

[11] There are many other graph algorithms, such as single source shortest path, depth first search, community detection, and triangle counting that use indicators for their executions. However, since existing graph engines do not provide them, we were not able to include them in our experiments.

6.2 Evaluation of proposed techniques

In this section, we verify the effectiveness of the techniques applied to RealGraph: Hierarchical indicator (H), Block-based workload allocation (B), and efficient Data layout (D). We performed two algorithms — BFS as a representative algorithm for accessing *part of* nodes at each iteration and PageRank as a representative algorithm for accessing *all* nodes at each iteration — on the Yahoo dataset, the largest one in experimental datasets. We set the number of threads of RealGraph as 8. The memory size is limited to 16GiB for the executions of out-of-core algorithms.

Figure 10 shows the results. The *x*-axis represents RealGraph with each technique applied. In the *x*-axis, Null means RealGraph with none of the proposed techniques (i.e., employing the flat indicator and node-based workload allocation), and each alphabet represents RealGraph equipped with the technique referring to its initial. For example, HB indicates RealGraph with the hierarchical indicator (H) and the block-based workload allocation (B). Ω represents RealGraph with all techniques applicable to each algorithm. The *y*-axis represents the execution time. And now, we describe the result in a way that compares the performance of RealGraph including any proposed technique with that of Null.

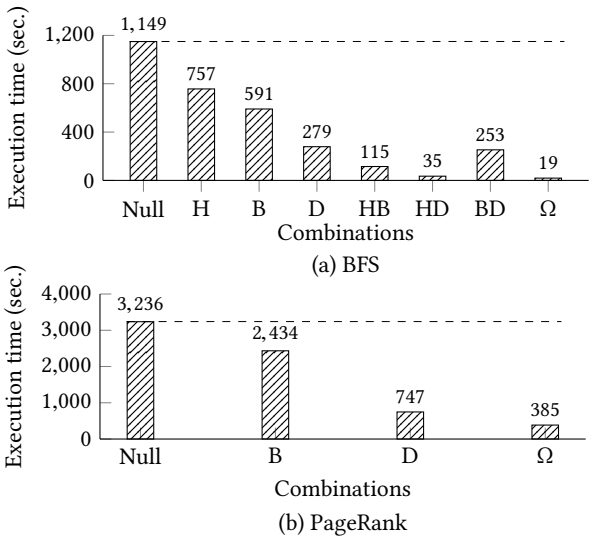

Figure 10: Performance of proposed techniques.

Figure 10-(a) shows the result of BFS. The hierarchical indicator (H) improves the performance about 1.5 times by reducing the scanning overhead, the block-based workload allocation (B) about 1.9 times by the nice load balancing, and the efficient data layout (D) about 4 times by increasing the degree of sequential access and the cache hit for attribute data and blocks, respectively.

When using both the hierarchical indicator and the block-based workload allocation (HB), the performance of RealGraph is improved 9.9 times. This is because the scanning overhead of the indicator at each iteration is reduced and the delay of the progress for the next iteration is prevented by even workload distribution. In addition, the main thread communicates with worker threads

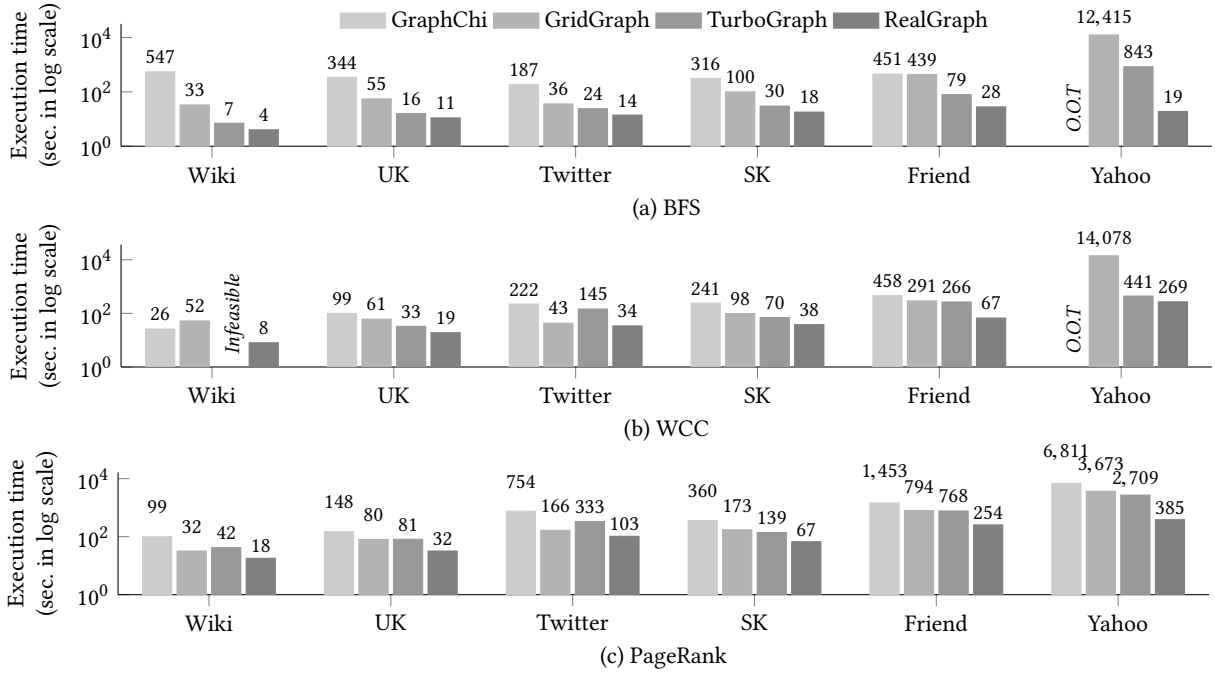

Figure 11: Performance comparison with graph engines with the same model.

per each block (rather than each node), which reduces the communication overhead. This further improves the performance of RealGraph.

The synergy is greater when combining the hierarchical indicator and the efficient data layout (HD). As mentioned in Section 3.2, the performance of RealGraph is improved by the efficient data layout when handling attribute data and graph data. Moreover, the number of ranges to be scanned in the current indicator decreases when efficient data layout is employed, because the bits corresponding to nodes to be processed tend to be clustered in a few ranges with efficient data layout. The cache hit ratio in the next indicator is also increased, because the bits in the same or adjacent ranges are likely to be set. This results in the *dramatic* performance improvement of RealGraph by 32.8 times.

The combination of the block-based workload allocation and the efficient data layout (BD) also shows synergy, improving the performance of RealGraph by 4.4 times. The magnitude, however, is somewhat smaller than the previous two cases (HB and HD). The reason is as follows. The efficient data layout aggregates the nodes to be processed into one or adjacent blocks. If an iteration processes a small number of nodes, the number of blocks to be processed may be less than the number of threads in RealGraph. Since each thread processes a block by the block-based workload allocation, all the given threads could have not been fully utilized in this case.

Finally, the combination of all three technologies (Ω) enhances the performance of RealGraph by 60 times.

Figure 10-(b) shows the results of PageRank. In PageRank where the indicator is not used, we could apply the block-based workload allocation (B) and the efficient data layout (D) to RealGraph. Each technique improves the performance by 1.3 and 4.3 times, respectively. The performance of RealGraph is improved 8.4 times when using both techniques together (Ω). The efficient data layout increases the probability that the nodes with sequential node IDs and the nodes connected to them take adjacent node IDs. If these nodes are processed by a single thread, the cache hit ratio for attribute data can be further increased. Because the block-based workload allocation assigns each thread the nodes in a single block with sequential node IDs, the performance is further improved by the synergy between the block-based allocation and the efficient data layout.

Through above experiments, we showed that each technique of RealGraph is effective, and that their combinations create a significant synergy effect. We also demonstrated that these techniques are all effective regardless of the type of graph algorithms.

6.3 Comparisons with single-machine based graph engines

In this experiment, we evaluate the performance of RealGraph by comparing it with existing single-machine based graph engines. Section 6.3.1 evaluates the performance of the single-machine based graph engines following the *same model* as RealGraph. Section 6.3.2 evaluates the performance of the single-machine based graph engines adopting *different models* from RealGraph. In these two sections, we used BFS, WCC, and PageRank, which are commonly provided by all graph engines. Section 6.3.3 shows the performance of other graph algorithms provided by different graph engines such as BC, RWR, BP, and SpMV.

6.3.1 Single-machine based graph engines with the same model. We employed TurboGraph [25], GridGraph [30], and GraphChi [14], all of which follow the same model as RealGraph. They adopt the vertex-centric model and the external-memory model. Since

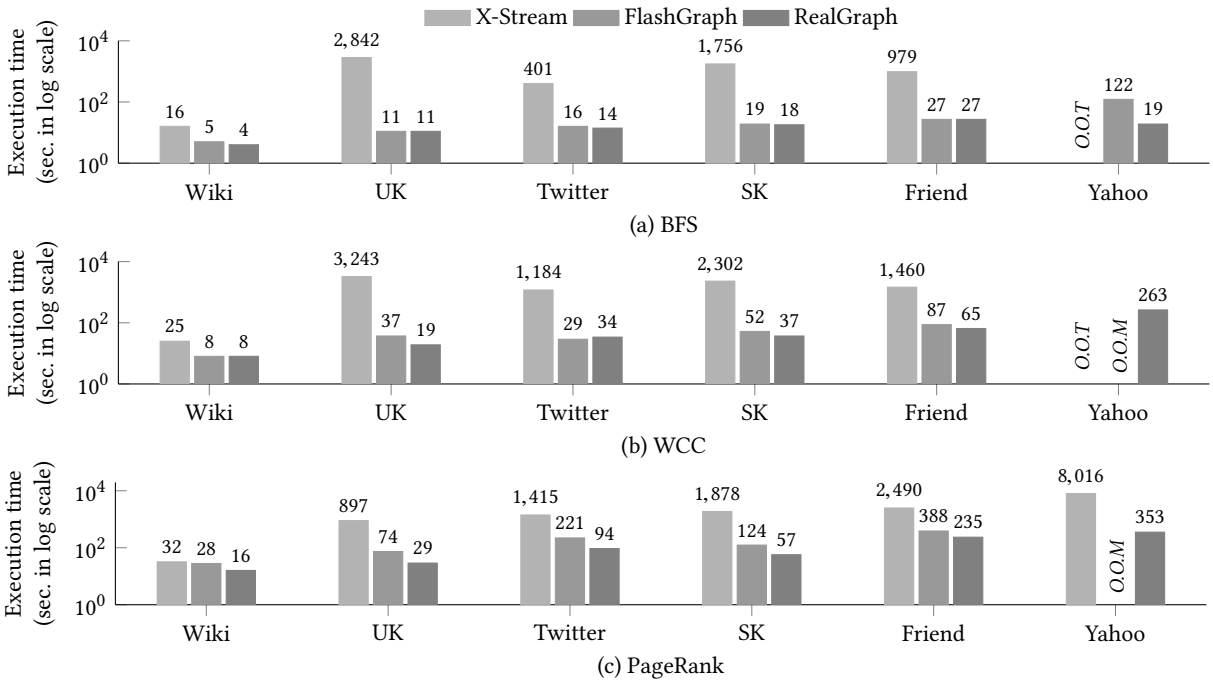

Figure 12: Performance comparison with graph engines with the different models.

TurboGraph fixed the number of threads identically as six, we set the number of threads of all the graph engines to six.

Figure 11 shows the results. The x-axis represents the graph datasets, and the y-axis represents the execution time in log scale. The experimental results show that overall execution time is in order of GraphChi, GridGraph, TurboGraph, and RealGraph. RealGraph provides the best performance *universally* for all graph algorithms and for all datasets. In particular, RealGraph is better than TurboGraph, the best performer among existing ones, up to about 44 times in the best case (i.e., BFS on the Yahoo dataset). We note that TurboGraph does not work in the Wiki dataset (i.e., infeasible), and GraphChi could not get the results of BFS and WCC on the Yahoo dataset within 24 hours (i.e., out of time, O.O.T).

6.3.2 Single-machine based graph engines with different models. In this experiment, we employed two graph engines with different models from that of RealGraph: FlashGraph [29] and X-Stream [21]. FlashGraph adopts the vertex-centric model and the semi-external memory model. X-Stream adopts the edge-centric model and the external-memory model. The number of threads was set identically as 8. We performed BFS, WCC, and PageRank, as in the previous experiments.

Figure 12 shows the experimental results. The x-axis represents the graph datasets, and the y-axis represents the execution time in log scale. RealGraph has the same or better performance than FlashGraph except for WCC on the Twitter dataset. In particular, the performance of RealGraph is up to six times better than that of FlashGraph (i.e., BFS on the Yahoo dataset). Since the graph engine with the semi-external memory model does not work if whole attribute data cannot be loaded into the given memory, FlashGraph could not perform PageRank and WCC on the Yahoo dataset (i.e., out of memory, O.O.M), while RealGraph performs all the algorithms.

Compared with X-Stream, RealGraph has always better performance for all graph algorithms and all datasets. In the case of PageRank on the Yahoo dataset, RealGraph outperforms X-Stream up to 250 times. Since the graph engines with the edge-centric model sequentially read the entire graph data at each iteration, it is inefficient for the algorithms that need to process only a few nodes at each iteration such as BFS and WCC. X-Stream does not work on BFS and WCC on the Yahoo dataset within 24 hours (i.e., O.O.T), while RealGraph works in maximum 360 seconds. These results show that RealGraph has superior scalability and better performance compared to two graph engines with different models.

6.3.3 Performance on other graph algorithms. This experiment shows that RealGraph is effective for other graph algorithms provided by different graph engines. BC is provided by RealGraph, GraphChi, and X-Stream; RWR is provided by RealGraph, FlashGraph, and X-Stream; BP is provided by RealGraph, GridGraph, and X-Stream; SpMV is provided by RealGraph, GraphChi, and X-Stream. TurboGraph does not provide any additional graph algorithms except for the common graph algorithms (i.e., BFS, PageRank,

Table 2: Execution times of other graph algorithms

Graph engines	Graph algorithms			
	BC	RWR	BP	SpMV
RealGraph	125.7	313.9	3017.3	112.2
TurboGraph	-	-	-	-
GridGraph	-	-	-	226.9
GraphChi	-	6082.6	8418.3	-
FlashGraph	O.O.M	-	-	-
X-Stream	O.O.T	8016.3	O.O.M	538.3

(Execution time unit: sec.)

Table 3: Performance comparison of RealGraph with distributed-system based graph engines

Graph engines	Graph algorithms		System environment			
	WCC	PageRank	Machine	#Cores	Mem. Size	Storage
RealGraph	34	94	Single PC with Intel i7	4	16G	SSD
PowerGraph	-	72	64 Amazon EC2 cc1.4xlarge	8	23G	HDD
GraphLab	244	249	16 Amazon EC2 m2.4xlarge	8	68G	HDD
GraphX	251	419	16 Amazon EC2 m2.4xlarge	8	68G	HDD
Giraph	200	596	16 Amazon EC2 m2.4xlarge	8	68G	HDD

(Execution time unit: sec.)

and WCC). The number of threads of all graph engines was identically set as 8. We performed the algorithms on the largest Yahoo dataset.

Table 2 shows the execution times of the algorithms on the graph engines. The hyphen (-) means that the execution time is not measured because the graph engine does not provide the corresponding algorithm. The experimental results show that RealGraph has the best performance for all graph algorithms, which proves that RealGraph are effective in various domains using real-world graphs.

6.4 Comparisons with distributed-system based graph engines

Finally, we compared RealGraph with four distributed-system based graph engines: PowerGraph [9], GraphLab [16], GraphX [26], and Giraph [1]. Since prior research [9, 26] has evaluated the performance of these four distributed-system based graph engines, we *quote* their results for comparison while measuring the performance of our RealGraph. Table 3 presents the performance comparison between RealGraph and distributed-system based graph engines for WCC and PageRank which are the only graph algorithms whose performances on the four distributed-system based graph engines are commonly reported [9, 26]) on the Twitter dataset. Table 3 also shows the system environment of each graph engine. where *Machine* indicates the number of and type of the machines, and for each machine, *#Cores* means the number of cores, *Mem. Size* does the size of memory, and *Storage* does the type of storage. The hyphen (-) means that no result is available.

Under typical configuration, the distributed-system based graph engine uses 16 or more machines, each of which has 8 cores and 20GiB or more memory. In other words, they have computing power *much* higher than RealGraph. Nonetheless, the result shows that RealGraph equipped with low computing power performs better than distributed-system based engines except PowerGraph.

When executing PageRank, PowerGraph employs 64 machines, each of which is equipped with 8 cores, while RealGraph uses a single machine with 4 cores. That is, PowerGraph uses 128 times more cores than RealGraph, notwithstanding other computing resources. The performance of PowerGraph, however, is only 25% better than that of RealGraph. Simply put, RealGraph achieves reasonable performance using much smaller computing resources.

The reasons are as follows. The graph algorithm is performed through multiple iterations, producing intermediate results proportional to the number of nodes at each iteration. To proceed to the next iteration, each machine in the distributed graph engine should maintain the same intermediate results. Thus, intermediate results should be synchronized through communication among machines, which may cause more overhead than processing of the graph algorithm [5]. We claim that RealGraph is more effective if multiple machines are not required and the given a graph can reside within a single machine.

7 CONCLUSIONS

Most real-world graphs follow the power-law degree distribution, which with a small number of hub nodes having a high degree and most nodes having a low degree. We have observed two important problems that occur when existing single-machine based graph engines process such real-world graphs: (1) inefficient scanning of the sparse indicator and (2) the delay for the progress of the iterations due to uneven workload distribution.

In order to solve these problems, we developed a new graph engine, RealGraph, designed on the basis of well-known database systems. We applied not only an efficient data layout with good data locality, but also following two techniques that efficiently process real-world graphs into RealGraph.

First, we proposed a *hierarchical indicator* where the higher-level indicator compresses a range of a lower-level indicator using a single bit. This helps to skip unnecessary ranges of each lower-level indicator by scanning each higher-level indicator, which provides efficient indicator scanning at every iteration.

Second, we devised a *block-based workload allocation* where RealGraph assigns blocks to threads. Since a block is a fixed-size standard I/O unit, the total size of objects in each block is almost the same. This prevents the delay for the progress of the next iteration.

Through extensive experiments, we verified the superiority of RealGraph. We first evaluated the effectiveness of each proposed technique. Each technology improves the performance of RealGraph, and the combination of technologies further improves its performance with synergy. For various graph algorithms and real-world graphs, RealGraph has the best performance in terms of speed and scalability compared to existing single-machine based and distributed-system based graph engines.

ACKNOWLEDGMENTS

This work was supported by (1) the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) (NRF-2017R1A2B3004581) and (2) Next-Generation Information Computing Development Program through NRF funded by MSIT (NRF-2017M3C4A7069440). Also, we appreciate Samsung Electronics' university program [Flash Solutions for Emerging Applications] that significantly helps train our lab. members.

REFERENCES

[1] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. In *Hadoop Summit*. 5–9.

[2] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology* 25, 2 (2001), 163–177.

[3] Michael J Carey, David J DeWitt, Michael J Franklin, Nancy E Hall, Mark L McAuliffe, Jeffrey F Naughton, Daniel T Schuh, Marvin H Solomon, CK Tan, Odysseas G Tsatalos, Seth J White, and Michael J Zwilling. 1994. Shoring up persistent applications. In *Proceedings of the ACM international conference on management of data (SIGMOD)*. 383–394.

[4] Michael J Carey, David J DeWitt, Joel E Richardson, and Eugene J Shekita. 1986. *Object and file management in the EXODUS extensible database system.* University of Wisconsin-Madison. Computer Sciences Department.

[5] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2014. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the international symposium on high-performance parallel and distributed computing (HPDC)*. 215–226.

[6] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. NXgraph: An efficient graph processing system on a single machine. In *Proceedings of the IEEE international conference on data engineering (ICDE)*. 409–420.

[7] H-T Chou, David J Dewitt, Randy H Katz, and Anthony C Klug. 1985. Design and implementation of the Wisconsin storage system. *Software: Practice and Experience* 15, 10 (1985), 943–962.

[8] Martin Erwig. 1992. Graph algorithms= iteration+ data structures?. In *International workshop on graph-theoretic concepts in computer science*. 277–292.

[9] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX symposium on operating systems design and implementation (OSDI)*. 17–30.

[10] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment* 8, 9 (2015), 950–961.

[11] Min-Hee Jang, Christos Faloutsos, Sang-Wook Kim, U Kang, and Jiwoon Ha. 2016. Pin-trust: Fast trust propagation exploiting positive, implicit, and negative information. In *Proceedings of the ACM international conference on information and knowledge management (CIKM)*. 629–638.

[12] Yong-Yeon Jo, Jiwon Hong, Myung-Hwan Jang, Jae-Geun Bang, and Sang-Wook Kim. 2016. Data Locality in graph engines: Implications and preliminary experimental results. In *Proceedings of the ACM international conference on information and knowledge management (CIKM)*. 1885–1888.

[13] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the ACM international conference on management of data (SIGMOD)*. 447–461.

[14] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a pc. In *Proceedings of the USENIX symposium on operating systems design and implementation (OSDI)*. 31–46.

[15] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data* 1, 1 (2007), 1–41.

[16] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

[17] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication. In *Proceedings of the USENIX annual technical conference (ATC)*. 195–207.

[18] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the ACM european conference on computer systems (EuroSys)*. 527–543.

[19] 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 ACM international conference on management of data (SIGMOD)*. 135–146.

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

[21] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM symposium on operating systems principles (SOSP)*. 472–488.

[22] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms*. Addison-wesley professional.

[23] Kenji Suzuki, Isao Horiba, and Noboru Sugie. 2003. Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding* 89, 1 (2003), 1–23.

[24] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. 2013. Asynchronous large-scale graph processing made easy. In *Proceedings of biennial conference on innovative data systems research (CIDR)*. 3–6.

[25] Wook-Shin, 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 ACM international conference on knowledge discovery and data mining (KDD)*. 77–85.

[26] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. GraphX: A resilient distributed graph system on spark. In *Proceedings of the international workshop on graph data management experiences and systems (GRADE)*. 1–6.

[27] Xintian Yang, Srinivasan Parthasarathy, and Ponnuswamy Sadayappan. 2011. Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining. *Proceedings of the VLDB Endowment* 4, 4 (2011), 231–242.

[28] Hilmi Yildirim and Mukkai S Krishnamoorthy. 2008. A random walk method for alleviating the sparsity problem in collaborative filtering. In *Proceedings of the ACM conference on recommender systems (RecSys)*. 131–138.

[29] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the USENIX conference on file and storage technologies (FAST)*. 45–58.

[30] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the USENIX annual technical conference (ATC)*. 375–386.