Redio: Accelerating Disk-Based Graph Processing by Reducing Disk I/Os

Chengwen Wu, Guangyan Zhang, Yang Wang, Xinyang Jiang, and Weimin Zheng

Abstract—Disk-based graph systems store part or all of graph data on external devices like hard drives or SSDs, achieving scalability without excessive hardware. However, massive expensive disk I/Os remain the major performance bottleneck of disk-based graph processing. In this paper, we propose Redio, a new approach to accelerating disk-based graph processing by reducing disk I/Os. First, Redio observes that it is feasible to accommodate all vertex states in main memory and this can eliminate almost all vertex-related disk I/Os. Second, Redio introduces a dynamic selective scheduling scheme to identify inactive edges in each iteration and skip them when and only when such skipping can bring performance benefit. To improve its effectiveness, Redioin corporates a compact edge storage to improve data locality and an indexed bitmap to minimize its memory and computation overheads. We have implemented a single-node prototype for Redio under the edge-centric computation model. Extensive experiments show that Redio consistently outperforms well-known edge-centric disk-based systems in all experiments, delivering an average speedup of $4.33\times$ on HDDs and $5.33\times$ on SSDs over the fastest among them (i.e., GridGraph). Experimental results also show that Redio delivers an average speedup of $3.13\times$ on HDDs and $1.28\times$ on SSDs over the fastest among representative vertex-centric disk-based systems (i.e., FlashGraph).

Index Terms—Disk I/O, edge-centric, graph processing, large graph, vertex-centric

1 Introduction

As graph analysis is gaining increasing attentions in both academia and industry, researchers have made a great effort to develop specialized systems for scalable graph computing. These systems can be broadly categorized into two types: in-memory systems that store all data in main memory [1], [2], [3], [4], [5], [6], [7], [8], and disk-based systems that store part or all of graph data on external storage devices [9], [10], [11], [12], [13]. In-memory systems require that the main memory is big enough to hold the whole graph, and thus they are often designed to work on a cluster of machines, and further require fast networks to achieve good performance. On the other hand, most of the disk-based graph systems are designed to work on a single machine, achieving scalability without excessive hardware. Previous works show that on many workloads, single-node disk-based graph systems can often achieve comparable or even better performance than distributed in-memory systems [9], [10], [11], [12].

At a high level, graph processing systems work in iterations: in eachiteration,the system needs to compute updates to vertices, usually based on updates in the last iteration, and applies those updates to vertex states at the end of the iteration.To do so,the system needs to first identify vertices updated inthe last iteration Blparcalled an *active set* in the rest of the paper). Then the system identifies all edges whose sources are in the active set: the destinations of these edges should be updated accordingly. For many algorithms, it is incorrect to apply updates in place on the vertices during the iteration (see an example in Section 2) and thus the system needs to first record those updates, either in memory or on disks, and apply them when all updates are generated. An algorithm terminates after a given number of iterations or when it converges.

In a disk-based graph system, all three operations—accessing vertices, accessing edges, and recording and applying updates—may incur disk I/Os. Such massive disk I/Os are the major performance bottleneck in disk-based systems [10], [14], even when using faster devices such as SSDs [10]. This paper proposes Redio, a new approach to reducing disk I/Os for graph processing.

To achieve our goal of reducing disk I/Os, we carefully analyze the I/O pattern of disk-based graph systems. Our analysis first reveals a significant discrepancy between vertex-related accesses and edge-related accesses: on the one hand, vertex-related accesses, including both direct accesses to vertices and accesses to record and apply updates, account for more than half of the total accesses. On the other hand, the number of vertices in a graph is usually one or two orders of magnitude smaller than that of edges. Such discrepancy suggests that it is beneficial to buffer all vertex data in main memory. Our analysis of real graph datasets and hardware trend has confirmed its feasibility. *Fully buffering vertices* allows our system to almost eliminate I/Os incurred for accessing vertices and for recording and applying updates.

- C. Wu, G. Zhang, and W. Zheng are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: drec.wu@foxmail.com, {gyzh, zwm-dcs}@tsinghua.edu.cn.
- Y. Wang is with the Department of Computer Science and Engineering, the Ohio State University, Columbus OH 43210. E-mail: wang.7564@osu.edu.
- X. Jiang is with the College of Computer Science and Technology, Jilin University, Changchun 130012, China. E-mail: jxy859@gmail.com.

Manuscript received 15 Oct. 2017; revised 9 Sept. 2018; accepted 2 Oct. 2018. Date of publication 10 Oct. 2018; date of current version 19 Feb. 2019.
(Corresponding author: Guangyan Zhang.)
Recommended for acceptance by M. Caccamo.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2875458

0018-9340 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Next we focused on edge-related I/Os. Our analysis, similar to previous works [11], [14], [15], has observed an opportunity for further I/O optimization: in each iteration, a number of edges can be inactive, because their sources are not in the active set, and thus it is not necessary to read them. Exploiting this opportunity, however, is not simple: while reading all edges [10] is certainly a waste of disk bandwidth, skipping all inactive edges will generate a large number of random I/Os and can significantly degrade disk performance.

Existing systems exploit this opportunity either by statically partitioning a graph before computation and skipping fully inactive partitions during computation [11], or by estimating the future active edge set and repartitioning graph data on disks [14], [15] during computation. Both these approaches may miss opportunities to skip edges or introduce additional overhead when their partitioning or estimation deviates from real I/O patterns. For example, if we use the static partitioning approach but a large partition contains only a few active edges, the system cannot skip any edges in the partition.

Fully buffering vertices allows our approach to have accurate information about which edges are active in each iteration. By utilizing such information, we propose *dynamic selective scheduling*: at the beginning of each iteration, our approach identifies all inactive edges and determines whether it is worthwhile to skip these edges. To realize this idea, we need to address two challenges:

The first challenge is how to maximize the benefit of skipping, given the fact that skipping may incur random I/Os. Our analysis reveals that both disk characteristics and graph data format may affect the answer: for disks, our system cleverly skips a number of consecutive inactive edges (called an inactive chunk) only if transferring these edges takes longer than skipping them as a whole. For graph data, if inactive edges are randomly scattered inside the raw graph file, then skipping them may incur a lot of random I/Os, which is certainly bad for performance. To improve locality for active edges, our system stores graph edges as an adjacent list [12] (i.e., place the out-edges of a vertex at adjacent positions), so that out-edges of an inactive vertex can be skipped as a whole.

The second challenge is how to minimize its memory and computation overheads. Before each iteration, our system needs to scan active vertices to identify active edges and compute inactive chunks accordingly. Our analysis shows that for algorithms which have only a few active edges in each iteration, minimizing scanning overhead is particularly important, because in this case, the scanning overhead is relatively high compared to I/O overhead. Based on this observation, we incorporate an indexed bitmap to address the challenge: the bitmap keeps track of which vertices are active and is divided into multiple partitions; the index identifies which partitions have at least one active vertex. Using a bitmap can minimize the memory consumption when there are many active vertices, and the index can allow our system to quickly identify and skip consecutive inactive vertices, which is particularly beneficial for algorithms which have only a few active edges in each iteration.

While part of the techniques used in this paper have been explored in previous works, the key contribution of this paper is the analysis of the I/O patterns of graph processing and the corresponding analysis-driven I/O optimizations, which include both invention of novel techniques (e.g., clever skipping) and appropriate application of existing techniques (e.g., vertex buffering, storing edges as an adjacent list).

We have implemented a single-node prototype for Redio under the edge-centric computation model (see Section 2), although we believe its ideas are also applicable to systems using vertex-centric model. Extensive experiments with three algorithms on five real-world graphs show that Redio consistently outperforms well-known edge-centric disk-based systems (e.g., X-Stream [10], and GridGraph [11]) in all experiments, delivering an average speedup of $4.33\times$ on HDDs and $5.33\times$ on SSDs over the fastest among them (i.e., GridGraph). Our detailed measurement shows that the total data size that Redio accessed from disk(s) is significantly smaller than that of GridGraph, which confirms the reason of Redio's performance improvement.

We also compared Redio with vertex-centric disk-based systems (e.g., GraphChi [9], and FlashGraph [12]). Experimental results show that Redio delivers an average speedup of $3.13\times$ on HDDs over the fastest among them (i.e., FlashGraph). Even when running on SSDs, which is the case FlashGraph is specifically optimized for, Redio can still outperform FlashGraph by 28 percent.

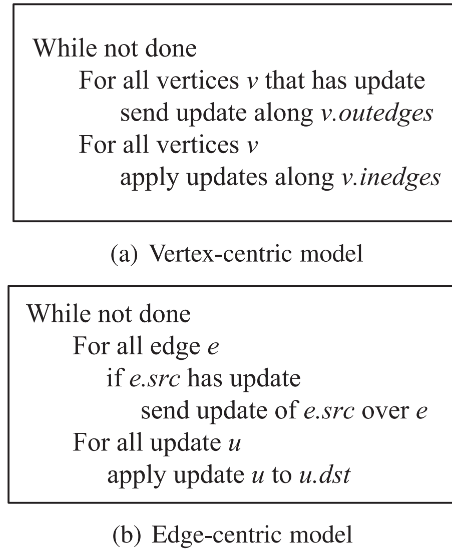
2 Background

Working Process Overview. We first present how disk-based graph systems work in general, using Breadth First Search (BFS) as an example.

At a high level, disk-based graph systems work in iterations: in each iteration, the system needs to compute updates to vertices, usually based on updates in the last iteration, and applies those updates at the end of iteration. Taking BFS as an example, in each iteration, the algorithm needs to 1) find vertices that are marked in last iteration (active set), 2) identify edges whose sources are in the active set, and 3) mark destinations of these edges, if they have not been marked yet. Note that for BFS, step 2) and step 3) cannot be overlapped, because marking vertices in step 3) may affect identifying edges in step 2) and overlapping them may violate the definition of BFS. To prevent such overlap in each iteration, graph systems usually first record updates to vertices in a separate space, and apply such updates at the end of the iteration.

As a result, disk-based graph systems may need to perform I/Os to access vertices, to access edges, and to record and apply updates. Based on their computation model, they can be classified into two major categories: vertex-centric and edge-centric, although recent works often incorporate ideas from both categories.

Vertex-centric Computation Model. Vertex-centric computation model iteratively executes a user-defined program over vertices of a graph [16]. In each iteration, it contains two main phases, *scatter* phase and *gather* phase. Fig. 1a demonstrates the pseudo-code of vertex-centric graph processing. In the scatter phase, an active vertex (i.e., a vertex that was updated in the last iteration) propagates its state to neighbors along its outgoing edges. The scatter phase performs sequential reads to vertices, random reads to edges, and random writes to updates. In the gather phase, a vertex accumulates updates from neighbors along its incoming edges to



(a) Vertex-centric model

(b) Edge-centric model

Fig. 1. Pseudo-codes of computation models used in disk-based systems.

recompute the vertex state. The gather phase performs random reads to edges, random reads to updates, and sequential writes to vertices.

Edge-Centric Computation Model. In the edge-centric computation model [10], each iteration also consists of a scatter phase followed by a gather phase. As shown in Fig. 1b, the scatter phase iterates over all edges and applies the user-defined scatter method to each edge. According to the state of the source vertex of a given edge, the scatter method computes and determines whether an update value needs to be sent to its destination vertex. If so, an update is produced. The scatter phase performs sequential reads to edges, random reads to vertices, and sequential writes to updates. The gather phase iterates over all updates and applies the user-defined gather method to each update. The gather method uses the value of an update to recompute the state of the destination vertex of the corresponding edge. The gather phase performs sequential reads to updates and random writes to vertices.

Comparison and Optimizations. As shown above, vertex-centric and edge-centric models have different access patterns and thus their performance depends on the algorithm. For example, vertex-centric model reads only active edges but may incur random I/Os, while edge-centric model reads all edges sequentially. As a result, vertex-centric model may perform better if the algorithm reads only a few edges in each iteration, and edge-centric model may perform better if the algorithm reads many edges.

Later works try to further optimize I/Os by taking ideas from both models: for example, GridGraph adopts the edge-centric model but it avoids reading all edges by dividing the whole graph into a 2D grid: in each iteration, if all edges in one cell is inactive, GridGraph will skip them; FlashGraph adopts the vertex-centric model but it makes I/Os as sequential as possible by re-ordering I/Os in its own local file system. Their experiments have shown that, despite these optimizations, *disk I/O is still the major bottleneck of disk-based graph systems*.

3 The Redio Approach

Redio accelerates disk-based graph processing by optimizing disk I/Os. First, we observe that, for most graphs we are aware of, it is feasible to accommodate all vertices in main memory, which leads to obvious performance benefit because vertices are usually accessed much more frequently than edges. Second, to strike a balance between avoiding reading inactive edges and incurring random I/Os, Redio introduces a dynamic selective scheduling scheme to cleverly skip inactive edges during computation, when and only when such skipping is beneficial for performance.

3.1 Fully Buffering Vertices in Main Memory

Our design is driven by the insights we gain from analyzing existing datasets and algorithms. From our analysis on five typical graph datasets and three typical algorithms, we draw some beneficial observations as follows.

Observation 1. In a disk-based graph processing system, the total number of vertex-related accesses, including both direct accesses to vertices and accesses to record and apply updates, is larger than that of edge-related accesses.

Analysis. In a vertex-centric system, an access to an edge is always associated with an access to an update (i.e., write an update in the scatter phase and read the update in the gather phase). In an edge-centric system, an access to an edge is always associated with at least one direct access to a vertex (i.e. access only the source vertex if the edge is inactive and access both source and destination vertices if the edge is active). Therefore, the total number of vertex-related accesses is larger than that of edge-related accesses.

Observation 2. For many graphs, it is feasible to hold all vertex state in memory.

Analysis. On the one hand, most real-world graphs follow the power law [17]. That is to say, the number of vertices in a graph is usually one or two orders of magnitude less than that of edges. Table 1 demonstrates the obvious difference

TABLE 1
Properties of Different Graphs

Dataset	Vertices	Edges	Average Degree	Vertex Set Size			
				PageRank1	PageRank2	BFS	WCC
Twitter [22]	61.6 M	1.47 B	23.86	469.81 MB	712.05 MB	249.58 MB	249.58 MB
UK [23]	106 M	3.74 B	35.28	815.33 MB	1.21 GB	433.15 MB	433.15 MB
Yahoo [24]	1.41B	6.64 B	4.71	10.53 GB	15.96 GB	5.59 GB	5.59 GB
GSH [25], [26]	988 M	33.87 B	34.28	7.36 GB	11.16 GB	3.91 GB	3.91 GB
Clueweb12 [25], [26]	978 M	42.57 B	43.51	7.29 GB	10.93 GB	3.87 GB	3.87 GB

Columns PageRank1, PageRank2, BFS (Breadth First Search), *and* WCC (Weakly Connected Component) *represent the size of vertex data when running the corresponding algorithm.*

between the numbers of vertices and edges. Large average degrees show that the number of edges in a graph is usually much larger than that of vertices. It also illustrates the sizes of vertex attribute subsets used in different algorithms. Running the *PageRank2* algorithm on the *Yahoo* dataset has the largest vertex state subset, whose size is 15.96 GB.

On the other hand, it is not uncommon for a today's server to have 32 GB of memory or more. Furthermore, the growth rate of memory capacity is similar to or higher than the growth rate of popular vertex entities: between 2011 and 2015, memory capacity per dollar has increased by about five times [18], while in the same period, number of active Facebook users has increased by about 2.3 times [19], and number of Google indexed pages has increased by about 4.2 times [20], [21].[1]

These two observations suggest that buffering all vertex data in main memory is both beneficial and feasible. Therefore, we decide to *buffer all vertex data in the memory to achieve high performance*, and to *store edge data on the storage for high scalability*.

Since Redio accommodates all vertex data in memory, its computation in each iteration can be simplified into a scatter-apply operation: in each iteration, Redio computes updates for each vertex, usually based on updates in the last iteration, and accumulates the generated updates onto a temporary copy of the corresponding vertex. At the end of the iteration, the system applies the values in the temporary copies directly to the corresponding vertices in main memory with the given accumulation operation. In Redio, such temporary copies are counted as vertex state and thus is stored in memory. Note that the size of vertex data in Table 1 already include size of temporary copies.

Benefits. Compared with traditional disk-based graph systems, Redio obtains obvious performance benefits.

- *Redio almost eliminates disk I/Os to access vertices*. To compute large graphs, the traditional disk-based computation systems often split a graph into multiple subgraphs. In each iteration, they read a subgraph's vertex states once at a time from storage and writes them to storage after computation of the subgraph. This requires to read and write the whole vertex once in an iteration. Conversely, Redio reads vertex states into memory before the first iteration and writes their final states to storage after termination.
- *Redio eliminates disk I/Os for maintaining updates*. The traditional disk-based computation model often requires disk I/Os to write temporary updates to storage and later to load these updates from storage for the apply operation. Redio accumulates and applies updates in memory, eliminating disk I/Os for maintaining update values.

Discussion. While fully buffering vertices is certainly not a novel idea [12], the key point of our analysis is that, since buffering all vertices becomes feasible even for large graphs, graph systems could take this as a premise and focus on edge-related accesses, as we will discuss next.

A system can certainly support partial buffering by either *mmaping* the vertex file into memory [11] or designing an internal cache, but it then needs to pay the overhead (e.g., index for in-memory vertices and metadata for replacement algorithm), even when the memory is enough. Moreover, if Redio needed to read a significant portion of vertices from the disk, it would not be efficient because vertex accesses would be mostly random accesses in Redio. Therefore, to answer the question about how to handle larger graphs: we do not think extending Redio to support partial buffering is a good idea. Instead, our answer is that if possible, the user can try to rent a machine with sufficient memory. If renting is not an option or if even the best machine cannot hold all vertices in memory, the user should consider distributed graph systems.

1. It is common that a company in the early phase can grow faster, but after reaching a certain scale, the growth rate usually drops.

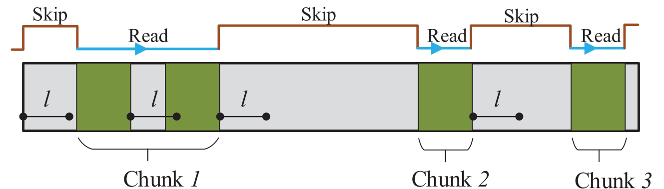


Fig. 2. Dynamic chunking for an iteration. Gray blocks represent inactive edges. If the interval between two adjacent active subblocks is smaller than the threshold, l, they are merged into a larger active subblock.

3.2 Dynamic Selective Scheduling

Edges and their edge attributes are stored on storage, and are loaded into main memory during computation. As shown above, in each iteration, it is possible that only part of the edges are active. For disk-based systems, reading only active edges may incur a large number of random I/Os, which is certainly bad for performance. Reading all edges, on the other hand, may waste disk bandwidth, especially when there are only a few active edges in each iteration. Redioin-troduces a dynamic selective scheduling scheme to skip inactive edges when and only when skipping is beneficial. To make it efficient, Redio incorporates a compact edge storage to improve data locality and an indexed bitmap to reduce computation and memory overheads.

Skip Inactive Edges Cleverly. In each iteration, Redio traverses over the vertex set to identify active edges: adjacent active edges become an active subblock. As shown in Fig. 2, if the total storage size of all inactive edges between two adjacent active subblocks is smaller than a threshold, l, they are merged into a larger active subblock. Finally, those active subblocks that cannot be merged further become active chunks. When streaming graph data in, Redio just loads those active chunks and skips inactive ones.

Here, the threshold, l, is set according to the performance feature of storage devices. The rule of choosing the l value for a hard drive is that the time of reading a l-length data block sequentially from a disk should be longer than one disk seek time. Therefore, we have $l/b_{seq} = t_{seek}$. Further, we get $l = b_{seq} \times t_{seek}$. In this way, Redio has the ability to skip inactive edges when and only when such skipping can bring performance benefit. The l value for an SSD is more difficult to estimate because of SSD's internal mechanism, and we set it based on experiments.

Compact Edge Storage. To improve the performance of dynamic selective scheduling, Redio applies a compact data format, similar to an adjacency list, to on-disk edge data to enhance data locality. This is because if a vertex becomes

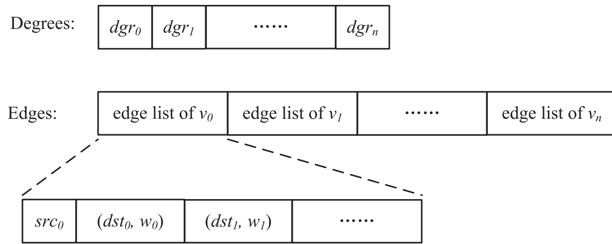


Fig. 3. Compact format of edge storage.

active, all its edges become active and thus storing them together can naturally create active subblocks.

Fig. 3 shows the data representation of a graph in Redio. In this format, edges with the same source vertex are put into adjacent positions. Then, we use an ID value to represent the source vertex of those edges with the same source vertex. Edge attributes of an edge come after its destination vertex. All the edges are ordered by their source vertex IDs. A graph in this format is stored in a single file. In addition, to know the number and the storage position of out-edges of a source vertex, Redio records the out-degree of each vertex in another file. It should be noted that as a byproduct, compact edge storage can reduce edge data by 34 to 50 percent compared to one that stores edges in the format of $(source, destination, value)$ (*cf.* Table 4), and thus may increase the speed of streaming in edges accordingly.

While such compact format has already been used in existing systems [12], Redio incorporates the same idea to improve the effectiveness of dynamic selective scheduling.

Indexed Bitmap. Dynamic selective scheduling needs to scan active vertices to identify active edges. Our analysis shows that depending on algorithms and graphs, the number of active vertices in each iteration can be drastically different. For example, some iterative algorithms, e.g., BFS and WCC, often access only a few active vertices in some iterations, especially when processing a sparse graph like the Yahoo graph. In these cases, it is important to minimize the time of scanning, because the I/O overhead is relatively low.

To achieve fast scanning with low memory overhead, we incorporate an indexed bitmap: the bitmap records whether a vertex is active or not and it is divided into multiple partitions with the size of p bits; the index, which is a much smaller bitmap, records whether a partition contains at least one active vertex. Given the number of vertices v, the memory overhead of an indexed bitmap is $(v + v/p)/8$ bytes.

During each iteration, Redio builds the indexed bitmap for the next iteration. When scanning the bitmap, Redio first scans the index and skips partitions without active vertices entirely. This can significantly reduce the scanning time for algorithms which have only a few active edges in each iteration, because the index will allow Redio to skip most inactive bits in the bitmap. When most vertices are active, however, the index cannot reduce the scanning time, but this is fine because this means that Redio needs to read many edges in this iteration and thus the scanning time is relatively short compared to the I/O time.

Benefits. Existing works either blindly partitions graphs [11], expecting some partitions will become inactive during computation, or try to estimate the active edges during computation and repartition graph data on disks [14], [15]. These approaches often miss opportunities to skip edges or

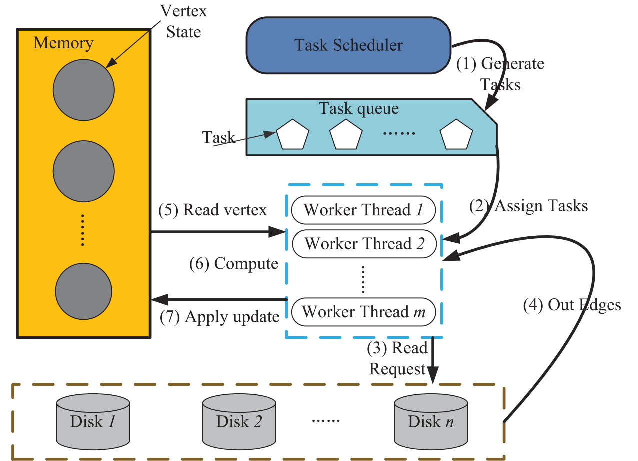


Fig. 4. The architecture and workflow of Redio. Vertex data are accommodated in main memory while edge data are stored on the storage.

introduce additional overhead when the actual I/O pattern deviates from expectation. For example, with the static partitioning approach, if a large partition contains a few active edges, then all inactive edges in the partition must be loaded. If partitions are over-small, the incurred random access to skip a partition may not be worth the data reduction we gain. Redio benefits from dynamic selective scheduling and thus can make more accurate decisions adaptively.

4 Implementation

We implement Redio by modifying the source code of GridGraph [11]. GridGraph is an open-sourced graph system, which has its own selective scheduling mechanism. Implementing Redio based on GridGraph makes it convenient to make a performance comparison between Redio and GridGraph. Redio preprocesses raw data once to generate compact edge data for all following computation. When executing an algorithm, Redio first creates arrays for vertex data, which will be buffered in main memory during the whole computation. After that, Redio performs computation for each iteration, accelerated by a dynamic selective scheduling scheme.

Preprocessing. Most raw graph data we are aware of are a long list of data items in the format of ($source$, $destination$, $value$). To generate compact edge data format, Redio performs data preprocessing by an external sort of records, whose keys are source vertices and values are destination vertices and edge attributes.

Vertex Data Buffering. If a vertex is attached with more than one attributes (e.g., PR value, degree, and partial sum for PageRank algorithm), Redio stores each attribute in a separate array. This design is based on the observation that, for most algorithms, only one or a subset of these values will be used at a time [27], and thus storing them separately can improve cache efficiency.

Redio first allocates an amount of memory for vertex buffering, while the rest of main memory are available for the OS page cache to buffer edge data. If the active edge data in this iteration can fit in the page cache, Redio uses the buffer I/O mode to read edge data. Otherwise, it uses the direct I/O mode to read edge data.

Computation. Fig. 4 demonstrates the architecture and main workflow of Redio. The overall graph processing is structured

as a loop, in which each iteration performs a scatter-apply operation. In an iteration, as shown in Fig. 4, the task scheduler first generates a number of tasks, each corresponding to a partition of edges (Step 1), and assigns them to different worker threads (Step 2). Then, each worker thread issues I/O requests to stream the corresponding active chunks in from the storage (Steps 3 and 4). For each active edge, the worker thread reads its source vertex value from main memory (Step 5), computes its update value and accumulates the generated update onto the temporary copy of the destination vertex (Step 6). At the end of this iteration, worker threads apply values in the temporary copies to destination vertices with the corresponding accumulation operation (Step 7).

Dynamic Selective Scheduling. At the beginning of an iteration, Redio identifies new active chunks and generates index information for active chunks. Such identification is based on the logic described in Section 3.2, and requires scanning the indexed bitmap, which is built when applying updates in the last iteration. To accelerate this procedure, Redio parallelizes its work by assigning a subset of vertices to each CPU core.

5 Evaluation

The goal of Redio is to enhance the performance of disk-based graph processing systems by optimizing their disk I/Os. To assess whether Redio achieves its goal, we evaluate the performance of Redio using various datasets and algorithms, and compare it to those of state-of-the-art graph processing systems. In particular, our evaluation answers the following questions:

- What is the performance gain of Redio over state-of-the-art graph processing systems? (*cf*. Section 5.2)
- Why can Redio achieve such performance gain? (*cf*. Section 5.3)
- How well does Redio's performance scale with the number of disks, number of CPU cores, and size of memory? (*cf*. Section 5.4)
- How long does Redio need to preprocess its data? (*cf*. Section 5.5)

5.1 Methodology

To answer those questions, we evaluate Redio with various datasets, algorithms, and hardware settings, and compare Redio to state-of-the-art graph systems:

Datasets. We use five real-world graphs as the datasets: Twitter [22], UK [23], Yahoo [24], GSH [25], [26], and Clueweb 12 [25], [26], in which Twitter represents social relationships and UK, Yahoo, GSH, and Clueweb12 represent links among web pages. Table 1 lists the number of vertices and edges in each graph. The Yahoo dataset is more sparse than others.

Algorithms. We run three popular algorithms—PageRank, Breadth First Search, and Weakly Connected Component (WCC)—on these graphs. PageRank first assigns a value to each vertex, then recomputes the value of each vertex based on the values sent from their neighbors in each iteration, and terminates after a given number of iterations (20 in our experiments) or when the mean difference between two iterations is small enough. In our experiments, we implemented both versions as PageRank1 [28] and PageRank2 [29]. BFS first puts a given vertex in the active vertex set, expands the active set to include its neighbors in each iteration, and terminates when no new vertex can be included. WCC first puts each vertex into a separate component, merges adjacent vertices into the same component in each iteration, and terminates when no new merging happens.

Comparison. We compare the performance of Redio to that of X-Stream, GridGraph, Graphchi, and FlashGraph, among them X-Stream and GridGraph are two edge-centric graph processing sytems and GraphChi and FlashGraph are two vertex-centric systems. Note that these systems all support buffering vertices when the capacity of memory is sufficient. For other system parameters, such as the partition parameter P in GridGraph, we have selected them with the approach recommended by the authors. The skipping thresholds of Redio are 1 MB for HDDs, and 64 KB for SSDs. For PageRank, we run PageRank2 for FlashGraph, and PageRank1 for other systems, because these are the algorithms used in the corresponding papers. We run both versions in Redio for comparison.

Hardware Setting. Our testbed uses a SuperMicro 2U server with two 4-core 2.50 GHz Intel XEON E5-2609 V2 processors and 32 GB DDR3 memory, running Ubuntu 14.04 with Linux kernel v3.13.0-32. The machine is equipped with eight 2TB HDDs, and two 500 GB SSDs, on them running EXT4 file systems. In the default setting, our experiments use all eight cores, 30 GB of memory (this includes both the memory used by the graph system and the memory used by OS file system buffer), a RAID-0 array of two HDDs, and a RAID-0 of two SSDs. In Section 5.4, we investigate how changing hardware resource affects the performance of Redio.

5.2 Performance

Our first set of experiments compares the performance of Redio to other disk-based systems with different datasets and algorithms. Tables 2 and 3 present the performance of different systems on HDDs and SSDs respectively. We category existing systems into vertex-centric and edge-centric ones. As shown in these tables, GridGraph and FlashGraph achieve the best performance respectively among edge-centric systems and vertex-centric ones. The last column of each type gives the speedup ratio of Redio over the best among edge-centric ones, and vertex-centric ones.

Redio consistently outperforms GridGraph in all experiments, with an average speedup of $4.33\times$ on HDDs and $5.33\times$ on SSDs. For each algorithm, we calculate a geometric mean of speedups of Redio over GridGraph on HDDs across five graphs. Redio provides an average speedup of 2.39×, 5.78×, and 5.90× for PageRank1, BFS, and WCC respectively. In particular, Redio's speedup is higher when running BFS and WCC on large graphs. This is because these algorithms take a large number of iterations to converge, while in most iterations, only a small number of edges are active. In this case, the benefits of Redio's dynamic selective scheduling are significant. Redio has the least speedup, 2.39× on average, when running PageRank1. This is because PageRank1 updates all vertices in each iteration and thus all edges are active. In this case, selective scheduling schemes of GridGraph and Redio do not take effect. Redio's improvement mainly comes from compact edge data, which reduces I/Os on streaming graph file.

TABLE 2
Runtime (in seconds) on 2 HDDs and Improvement of Redio Over Other Systems

	Redio	Edge-centric X-Stream	Edge-centric GridGraph	SpeedUp^e (× times)	Vertex-centric GraphChi	Vertex-centric FlashGraph	SpeedUp^v (× times)
Twitter							
PageRank1	213.85	2194.49	428.52	2.00	1833.47		
PageRank2	192.20					180.92	0.94
BFS	52.01	539.91	136.40	2.62	179.05	180.35	3.44
WCC	53.89	1777.3	140.13	2.60	1393.18	76.90	1.43
UK							
PageRank1	267.52	5424.28	735.53	2.75	2068.81		
PageRank2	204.62					224.08	1.10
BFS	143.21	9922.83	364.22	2.54	8243.35	703.94	4.92
WCC	142.76	15546.3	370.98	2.60	3505.99	221.44	1.55
Yahoo							
PageRank1	1637.68	10681.3	5812.25	3.55	10205.0		
PageRank2	1522.89					4994.38	3.28
BFS	668.05	-	19551.93	29.27	33625.2	3199.21	4.79
WCC	748.95	-	23225.57	31.01	-	6942.48	9.27
GSH							
PageRank1	7671.33	-	20356.54	2.65	-		
PageRank2	7625.81					74830.1	9.81
BFS	3550.23	-	21521.54	6.06	-	11666.9	3.29
WCC	2916.05	-	16175.91	5.55	-	19810.2	6.79
Clueweb12							
PageRank1	15891.73	-	23837.6	1.50	-		
PageRank2	15551.29					76583.6	4.92
BFS	10361.97	-	56569.71	5.46	-	12750.8	1.23
WCC	9281.83	-	56978.01	6.14	-	33233.3	3.58
GEOMEAN				4.33			3.13

"-": fails to get the result in 48 hours. SpeedUp^e is Redio's speedup over the best of edge-centric systems, while SpeedUp^v is Redio's speedup over the best of vertex-centric systems.

TABLE 3
Runtime (in seconds) on 2 SSDs and Improvement of Redio Over Other Systems

	Redio	Edge-centric X-Stream	Edge-centric GridGraph	SpeedUp^e (× times)	Vertex-centric GraphChi	Vertex-centric FlashGraph	SpeedUp^v (× times)
Twitter							
PageRank1	175.81	1010.60	209.92	1.19	1817.67		
PageRank2	155.44					114.71	0.74
BFS	13.54	254.15	26.98	1.99	161.6	14.41	1.06
WCC	15.00	826.582	31.66	2.11	1404.07	16.33	1.09
UK							
PageRank1	164.09	2611.44	273.42	1.67	1871.28		
PageRank2	102.19					102.72	1.01
BFS	25.15	4667.94	155.1	6.17	1996.41	34.06	1.35
WCC	37.52	7347.23	149.81	3.99	3317.76	103.53	2.76
Yahoo							
PageRank1	639.26	5803.13	1753.42	2.74	7556.26		
PageRank2	525.24					866.58	1.65
BFS	126.82	-	5562.17	43.86	16691.1	127.34	1.00
WCC	161.07	-	6851.41	42.54	-	529.27	3.29
GSH							
PageRank1	2918.63	*	7110.22	2.44	*		
PageRank2	2867.19					5533.78	1.93
BFS	763.61	*	6358.98	8.33	*	581.30	0.76
WCC	856.94	*	4949.57	5.78	*	1633.29	1.91
Clueweb12							
PageRank1	5151.24	*	10687.6	2.07	*		
PageRank2	5053.37					4369.55	0.86
BFS	1134.85	*	21410.23	18.86	*	734.61	0.64
WCC	1326.95	*	21677.21	16.34	*	2008.9	1.51
GEOMEAN				5.33			1.28

"-": fails to get the result in 48 hours, and "": fails to run due to the limited capacity of SSDs in our experiments. SpeedUp^e is Redio's speedup over the best of edge-centric systems, while SpeedUp^v is Redio's speedup over the best of vertex-centric systems.*

TABLE 4
Comparison of Edge Data Size between Binary Edge List and Redio's Compact Edge Storage

Dataset	Original	Compact	↓ %
Twitter	11 GB	5.8 GB	47.27%
UK	29 GB	15 GB	48.28%
Yahoo	50 GB	33 GB	34%
GSH	253 GB	134 GB	47.04%
Clueweb12	318 GB	162 GB	49.05%

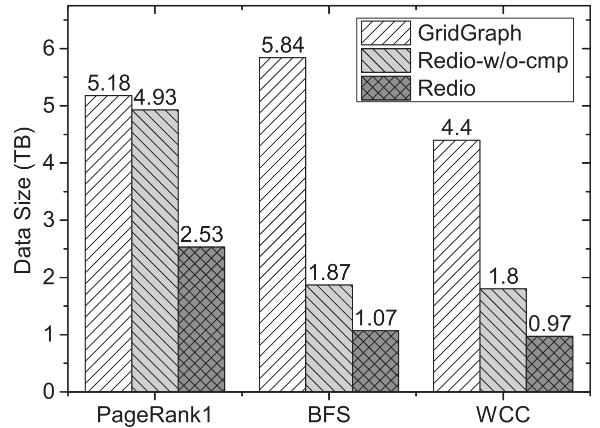
On two HDDs, Redio outperforms FlashGraph in all the experiments, except running PageRank2 on Twitter, with an average speedup of $3.13\times$. Even when running on SSDs, which is the case FlashGraph is specifically optimized for, Redio can still outperform FlashGraph by 28 percent. This is because reads of FlashGraph to edge data are random, although some I/Os are merged by the underlying file system specially designed for FlashGraph. However, it should be noted that because the edge-centric model and the vertex-centric model have intrinsic differences, the performance comparison of Redio to vertex-centric systems like FlashGraph may not be able to accurately measure the improvement of Redio's ideas: when applying Redio's ideas to a vertex-centric system, the actual improvement could be different. This set of experiments just serve a sanity check that Redio's performance is at least comparable to vertex-centric systems.

5.3 Disk I/O Analysis

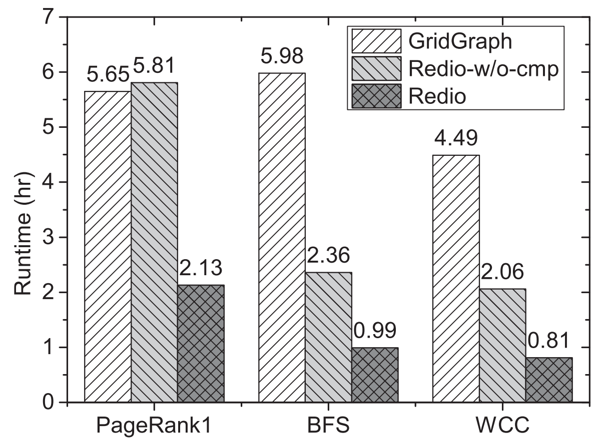
To answer why Redio can achieve such performance gain, our second set of experiments compares the total disk I/O size of Redio to that of GridGraph with the GSH graph, since GridGraph is the best system that follows the same model of Redio in our experiments.

Redio optimizes disk I/Os by using vertex buffering and dynamic selective scheduling. As shown in the analysis in Section 3.1, vertices' accesses account for more than half of all accesses in a disk-based system and buffering all vertices can completely eliminate these I/Os. Unfortunately, it is impossible to quantitatively measure such improvement in experiments because disabling or shrinking the buffer requires a significant re-design of Redio. Dynamic selective scheduling reduces I/Os in two ways: its clever skipping can skip inactive edges when there is a benefit to do so; its compact edge format, which stores edges as an adjacent list, can reduce the size of the graph compared to GridGraph's format, which uses binary edge list to store edge data. Next, we measure the data reduction Redio gains from compact edge format and dynamic selective scheduling.

Table 4 compares the size of Redio's compact edge file to the size of graph file used in GridGraph. As shown in the table, compaction can reduce graph size by 34 to 50 percent. This is because in these graphs, the number of vertices is usually much smaller than that of edges, indicating that many edges share same source vertices. Therefore, by compacting common source vertices IDs into one ID, Redio can greatly reduce graph size. Since all the graphs in the experiments are unweighted, Redio can gain a compaction ratio of up to 50 percent, while on weighted graphs, the compaction ratio may be smaller, unless weights can also be compressed.



(a) Data size accessed from the disk



(b) Runtime

Fig. 5. Disk I/O size and runtime of GridGraph, Redio-w/o-cmp (Redio without compacting edge data), and Redio on GSH graph.

To understand the effectiveness of dynamic selective scheduling, we compare the amount of data, accessed from the disk by Redio, to that of GridGraph, which has its own selective scheduling mechanism. Because GridGraph does not compact edge data, for fairness, we also run Redio without compaction, though out-edges of a vertex are still placed at adjacent locations.

We collected disk I/O size with the `iostat` tool, and showed data size accessed from the disk in Fig. 5a. For BFS and WCC, the total amount of data Redio accesses from the disk is about one fifth (i.e., 18 and 22 percent) of that of GridGraph: this explains the over five times speedup Redio gains over GridGraph. Even if Redio does not incorporates compaction, Redio-w/o-cmp is still over 2.18× more effective. For PageRank1, since the algorithm accesses all edges in every iteration, dynamic selective scheduling does not take effect. As a result, Redio-w/o-cmp and GridGraph have almost the same accessed data size and runtime. However, compacting edge data reduces data accessed by over 50 percent. This once again explains the $2.65\times$ speedup Redio gains over GridGraph.

The inefficiency of selective scheduling in GridGraph comes from the imbalance among the partitions GridGraph generates for scheduling: its 2-dimensional edge partitioning has produced some over-large partitions and many over-small partitions. For example, in all the 512×512 partitions GridGraph generated for the GSH graph, there are

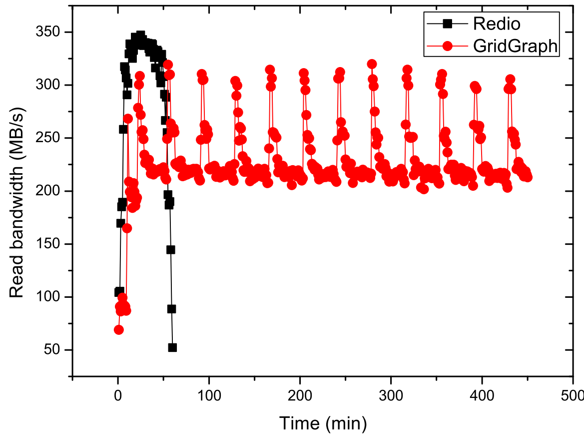


Fig. 6. Comparison of read bandwidth between Redio and GridGraph.

77 partitions larger than 500 MB, and 2,334 partitions smaller than 10 KB. Since a partition is the minimal unit for scheduling, GridGraph will load a partition, as long as one of its edges is active, even if all its other edges are inactive. Therefore, over-large partitions tends to miss many opportunities to skip inactive edges. Moreover, over-small partitions will incur many random accesses to edge data. Redio, on the other hand, exploits dynamic selective scheduling to skip edges when skipping can bring performance benefit. On the GSH graph, Redio can skip more edges than GridGraph, and introduces very low random access overhead.

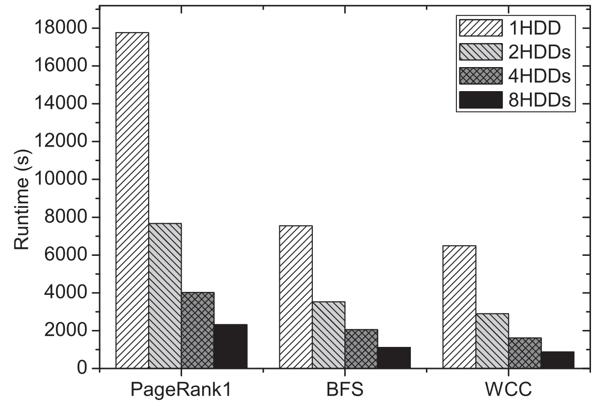
Fig. 6 compares Redio and GridGraph in read bandwidth during the execution of BFS on GSH graph. The results are reported by the *iostat* tool, with the sampling interval at 60 seconds. Since GridGraph buffers vertex data when the memory capacity is sufficient, the write bandwidth of Redio and GridGraph is almost 0, so we do not show the comparison of the write bandwidth here. From Fig. 6, we can see that Redio's read throughput is obviously higher than that of GridGraph. This indicates that Redio is more I/O efficient, which enhances the performance of graph processing.

5.4 Scalability

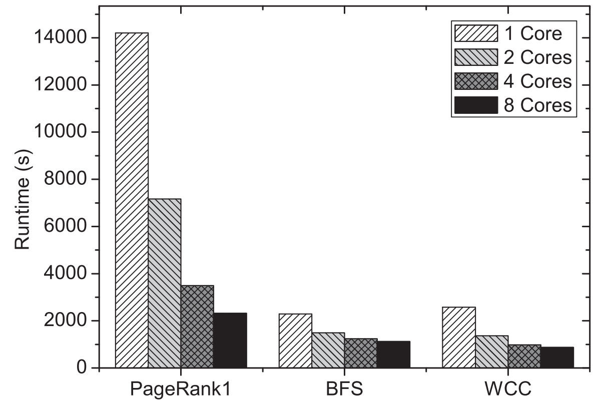
We evaluate the scalability of Redio in terms of number of disks, number of CPU cores, and size of memory.

Scalability in Disk Counts. Fig. 7a presents how performance scales with the number of hard disks. The disks are organized as a RAID-0 array. We choose GSH as the dataset, because it is large enough and thus tends to incur more disk I/Os. We use eight cores and 30 GB of memory in this experiment. As shown in the figure, Redio's performance scales linearly with disk bandwidth.

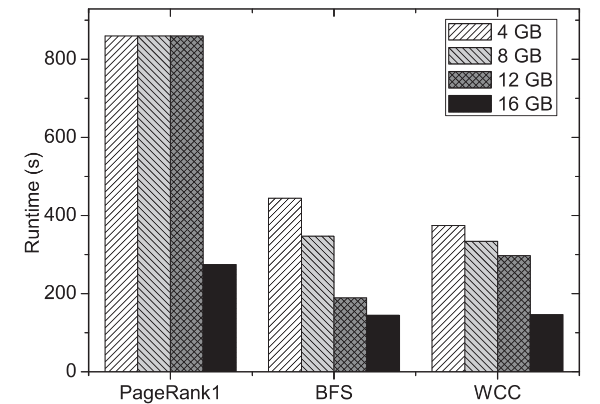
Scalability in Core Counts. Fig. 7b shows how performance scales with the number of cores. We use eight HDDs in this experiment to avoid I/O bottleneck. We again choose GSH as the dataset because algorithms running on this dataset has long execution time. As shown in the figure, the performance of PageRank1 scales linearly with the number of CPU cores, while for BFS and WCC, the performance enhancement is smaller when the number of CPU cores is increased to 4 or more. The reason is that 8 HDDs provide sufficient I/O bandwidth, so that when more cores are used, Redio can get better performance, but Redio needs to pay a synchronization overhead for each iteration (e.g., schedule work at the beginning and wait for barrier at the end): PageRank1 requires more CPU resources when doing computation and will load all edge data, while BFS and WCC require reading a comparatively small part of the graph data in each iteration and cost less CPU resources, so the overhead becomes significant and it limits the CPU scalability of Redio.



(a) Scalability in disk counts (GSH graph)



(b) Scalability in core counts (GSH graph)



(c) Scalability in memory size (UK graph)

Fig. 7. Scalability with hardware resource.

Comparing Fig. 7b to Fig. 7a, one can observe that in Redio, both the number of disks and the number of cores have impacts on performance. This demonstrates the effectiveness of Redio: comparing to previous systems that spend most time on I/Os, Redio has reached a point that I/O and CPU overheads are comparable.

Scalability in Memory Size. In order to measure how the performance of Redio scales with memory size, we change the memory size seen by the system by modifying the booting parameter—"mem". To provide a fair performance

comparison, we ensure that Redio and the counterparts use the same amount of main memory. Fig. 7c shows how performance scales with memory size. We use two HDDs and eight cores in this experiment. We choose a small dataset UK for this experiment because we hope to explore both the case that memory size is close to dataset size and the case that memory size is much smaller than dataset size. While in general, Redio can provide better performance with more memory, such performance improvement does not grow linearly with memory size. For example, when running PageRank1 on UK, increasing memory size from 4 GB to 12 GB does not improve system performance, but increasing it from 12 GB to 16 GB can provide a significant improvement. This is actually a classic behavior of a buffering system: Redio explicitly uses memory to buffer all vertices. Besides that, OS buffers edge data. If OS buffer is smaller than a threshold, its buffered edges will be evicted before they can be used again and thus buffering has no benefit. We will investigate how to buffer edges efficiently in the future.

5.5 Preprocessing Time

Our last set of experiments compares the preprocessing time of different systems. Most graph systems need such preprocessing phase to convert raw graph data into the format that can be utilized by the graph system. Since preprocessing only needs to be performed once for all future computations, its overhead will be amortized if the graph is used in computation multiple times.

X-Stream's preprocessing time is almost negligible because it actually merges preprocessing into the first iteration of computation. Even so, because of X-Stream's long computation time (see Table 2), when considering preprocessing time and computing time together, Redio is still significantly faster, even if preprocessing must be executed once for every computation.

Table 5 shows the preprocessing time of GraphChi, GridGraph, FlashGraph and Redio on 2 HDDs. One can observe that Redio's preprocessing time is less than those of other systems. Our investigation shows that such difference is due to various reasons. For example, Redio's assumption that all vertices can be buffered allows more efficient external sorting. FlashGraph's raw input is in text format, which needs more time to parse than one in binary format. Since some of these reasons are not fair across different systems (e.g., FlashGraph's text input), we cannot draw conclusions on preprocessing times but simply use this experiment to show that Redio will not incur a long preprocessing time that negates its benefit in computation.

TABLE 5
Preprocessing Time (in seconds) for GraphChi, GridGraph, FlashGraph and Redio

	Twitter	UK	Yahoo	GSH	Clueweb12
GraphChi	303	1,009	2,839	11,203	16,851
GridGraph	187	756	2,935	8,303	18,644
FlashGraph	1,220	3,427	6,235	23,966	24,570
Redio	114	478	911	4,563	6,652

6 Related Work

So far, a number of graph systems have been developed, including in-memory systems and disk-based ones, which inspire the design of Redio in several aspects.

6.1 In-memory Graph Systems

While in-memory systems usually require a cluster of machines to process large graphs, single-node systems are also developed to handle smaller graphs efficiently.

Single-node Systems. Ligra [2] is a graph processing framework, specifically optimized for graph traversal algorithms. Ligra+ [3] compresses graph representation to save memory space, and hence improve scalability. Galois [4] is a lightweight infrastructure for graph analytics, on which existing graph systems can be implemented efficiently. GRACE [5] provides an asynchronous computation model to speed up processing. Polymer [8] optimizes in-memory graph systems on multicore by NUMA-aware designs. It differentially places various kinds of data to minimize remote accesses. The obvious limitation of in-memory graph systems on a single server is their poor scalability: they can process only small-scale or at most medium-scale graphs. Redio buffers all the vertex data in the memory to achieve high performance, and stores edge data on the storage for high scalability.

Distributed Systems. Distributed graph processing systems can process large-scale graphs efficiently. Pregel [16] exploits bulk-synchronous processing (BSP) [30] and runs algorithms in a vertex-centric way. Giraph [31] and Hama [32] are the open-source clones of Pregel. GraphLab [33] processes large graphs asynchronously, making it a good fit for machine learning and data mining graph algorithms. PowerGraph [6] addresses the problems of graph placement and applies efficient optimization for graphs that follow the power law [17]. PEGASUS [34] is built on MapReduce [35], and expresses graph algorithms in the form of sparse matrix-vector multiplication. GraphX [7] is built upon Spark [36], and supports Pregel and GraphLab abstractions by using dataflow operators provided by Spark. Trinity [37] extracts the online and offline graph access patterns, and optimizes memory access and message passing. Gemini [38] applies multiple optimizations targeting computation performance to build scalability on top of efficiency. Those optimizations include a sparse-dense signal-slot abstraction, a chunk-based partitioning scheme, a dual representation scheme, NUMA-aware subpartitioning, plus locality-aware chunking and fine-grained work-stealing.

6.2 Disk-based Graph Systems

According to computation styles, disk-based graph systems are classified into vertex-centric and edge-centric.

Vertex-centric Systems. To reduce random I/Os, GraphChi [9] partitions graph into equal-sized shards and introduces a parallel sliding window mechanism to process each shard in turn. TurboGraph [39] is designed for SSDs, and extracts more parallelism than GraphChi by overlapping CPU processing and disk I/Os. VENUS [40] splits a graph into g-shards and the corresponding v-shards, which enables streamlined processing to overlap disk I/Os and computation. FlashGraph [12] stores vertex states in memory and edges on SSDs. It uses a user-space file system to merge I/Os and improve the locality of page cache. Although Redio also buffers vertex in memory, Redio introduces optimization techniques for hard disks and SSDs.

Edge-Centric Systems. X-Stream [10] reads all the edges in each iteration, which results in massive disk I/Os. Chaos [13] is a distributed graph system, which utilizes the disks of the computing machines to scale to very large graphs. GridGraph [11] constructs 2-dimensional edge blocks and adopts selective scheduling to skip unnecessary edge grids. However, its scheduling granularity is a grid partition, whose size is sometimes over-large or over-small, which will lead to reading many unnecessary edges or incur random I/Os respectively. Redio's dynamic selective scheduling can skip edges when the incurred random I/Os are worth the amount of data reduction.

Optimizations. FastBFS [15] optimizes X-Stream for the BFS algorithm by trimming the edges that have been accessed in the last iteration. Vora et al. [14] mitigate edge I/Os for disk-based systems by eliminating edges with zero new contribution to create dynamic partitions before every iteration. These works reshape partitions in each iteration, which incurs runtime overhead and extra I/O overhead. Redio's dynamic selective scheduling does not need extra disk I/Os since Redio does not perform physical data partitioning. Furthermore, since these works' reshaping is based on estimation of future workload, their performance depends on the accuracy of estimation, while Redio's dynamic selective scheduling is based on accurate runtime information of each iteration.

7 Conclusion

In this paper, we carefully analyze I/O patterns and propose Redio, a new approach to reducing I/Os for disk-based graph processing. First, fully buffering vertices allows Redio to eliminate almost all vertex-related disk I/Os. Second, Redio strikes a balance between avoiding reading inactive edges and incurring random I/Os, with low computation and memory overheads. As a result, the total data size that Redio accessed from storage is reduced significantly.

We have implemented a single-node prototype for Redio under the edge-centric computation model. Extensive experiments show that Redio consistently outperforms well-known edge-centric disk-based systems in all experiments, delivering an average speedup of $4.33\times$ on HDDs and $5.33\times$ on SSDs over the fastest among them (i.e., GridGraph). We also compared Redio with representative vertex-centric disk-based systems. Experimental results show that Redio delivers an average speedup of $3.13\times$ on HDDs and $1.28\times$ on SSDs over the fastest among them (i.e., FlashGraph).

Acknowledgments

This work was supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2014CB340402, the National Natural Science Foundation of China under Grant No. 61672315.

References

[1] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 301–316. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentatio n/zhu

[2] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 135–146.

[3] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *Proc. Data Compression Conf.*, 2015, pp. 403–412. [Online]. Available: http://dx.doi.org/10.1109/DCC.2015.8

[4] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 456–471.

[5] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *Proc. 6th Biennial Conf. Innovative Data Syst. Res.*, Jan. 2013, pp. 1–12.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.

[7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 599–613.

[8] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, 2015, pp. 183–193.

[9] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 31–46.

[10] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 472–488.

[11] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 375–386.

[12] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 45–58.

[13] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proc. 25th Symp. Operating Syst. Principles*, 2015, pp. 410–424.

[14] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic I/O optimization for disk-based graph processing," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2016, pp. 507–522. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/vora

[15] S. Cheng, G. Zhang, J. Shu, Q. Hu, and W. Zheng, "FastBFS: Fast breadth-first graph search on a single server," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 303–312.

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[17] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *Proc. Conf. Appl. Technol. Archit. Protocols Comput. Commun.*, 1999, pp. 251–262.

[18] M. T. ever-predictable DRAM path, [Online]. Available: http://www.pwc.com/gx/en/industries/technology/mobile-innovation/dram-memory-device-innovation.html, last visited in Oct. 2017.

[19] N. of monthly active Facebook users worldwide as of 2nd quarter 2016 (in millions), [Online]. Available: https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/, last visited in Oct. 2017.

[20] T. N. of Pages Indexed by Google, [Online]. Available: http://www.statisticbrain.com/total-number-of-pages-indexed-by-google/, last visited in Oct. 2017.

[21] T. size of the World Wide Web (The Internet), [Online]. Available: http://www.worldwidewebsize.com/, last visited in Oct. 2017.

[22] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 591–600.

[23] P. Boldi, M. Santini, and S. Vigna, "A large time-aware web graph," *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, Nov. 2008.

[24] YAHOO, "Yahoo! altavista web page hyperlink connectivity graph, circa 2002, " [Online]. Available: http://webscope.sandbox.yahoo.com/, last visited in Oct. 2017.

[25] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. 13th Int. World Wide Web Conf.*, 2004, pp. 595–601.

[26] P. Boldi, R. M., M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proc. 20th Int. Conf. World Wide Web*, 2011, pp. 587–596.

[27] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng, "Exploring the hidden dimension in graph processing," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 285–300. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhang-mingxing

[28] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, pp. 107–117, 1998.

[29] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 8, pp. 2091–2100, Aug. 2014.

[30] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[31] A. Giraph, [Online]. Available: http://giraph.apache.org/, last visited in Oct. 2017.

[32] A. Hama, [Online]. Available: https://hama.apache.org/, last visited in Oct. 2017.

[33] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein,"Distributed graphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[34] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proc. 9th IEEE Int. Conf. Data Mining*, 2009, pp. 229–238.

[35] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

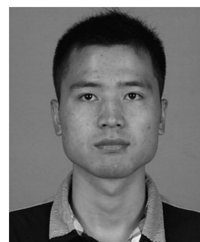
[36] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 15–28.

[37] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 505–516. [Online]. Available: http://doi.acm.org/10.1145/2463676.2467799

[38] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 301–316. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

[39] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 77–85.

[40] J. Cheng, Q. Liu, Z. Li, W. Fan, J. Lui, and C. He, "VENUS: Vertex-centric streamlined graph computation on a single PC," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 1131–1142.



Chengwen Wu received the bachelor's degree in computer science from the Beijing University of Posts and Telecommunications, in 2014. He is currently working toward the master's degree in the Department of Computer Science and Technology, Tsinghua University. His current research interest is in big data processing and network storage.

Guangyan Zhang received the bachelor's and master's degrees in computer science from Jilin University, in 2000 and 2003, respectively, and the doctorate degree in computer science and technology from Tsinghua University, in 2008. He is now an associate professor and PhD advisor in the Department of Computer Science and Technology, Tsinghua University. His current research interests include big data computing, network storage, and distributed systems. He is a professional member of the ACM.

Yang Wang received the bachelor's and master's degrees in computer science and technology from Tsinghua University, in 2005 and 2008, respectively, and the doctorate degree in computer science from the University of Texas at Ausitn, in 2014. He is now an assistant professor in the Department of Computer Science and Engineering, Ohio State University. His current research interests include distributed systems, fault tolerance, and scalability.

Xinyang Jiang received the bachelor's degree in software engineering from Jilin University, in 2014. He is currently working toward the master's degree in the College of Computer Science and Technology, Jilin University. His current research interest is in big data processing and network storage.

Weimin Zheng received the master's degree from Tsinghua University, in 1982. He is now a professor with the Department of Computer Science and Technology, Tsinghua University. His research covers distributed computing, compiler techniques, and network storage.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.