A Structure-aware Storage Optimization for Out-of-Core Concurrent Graph Processing

Xiaofei Liao, *Member, IEEE,* Jin Zhao, Yu Zhang, *Member, IEEE,* Bingsheng He, *Member, IEEE,* Ligang He, *Member, IEEE,* Hai Jin, *Fellow, IEEE,* Lin Gu, *Member, IEEE*

Abstract—With the huge demand for graph analytics in many real-world applications, massive iterative graph processing jobs are concurrently performed on the same graphs and suffer from significant high data access cost. To lower the data access cost toward high performance, several out-of-core concurrent graph processing solutions are recently designed to handle concurrent jobs by enabling these jobs to share the accesses of the same graph data. However, the set of active vertices in each partition are usually different for various concurrent jobs and also evolve with time, where some high-degree ones (or called *hub-vertices*) of these active vertices require more iterations to converge due to the power-law property of real-world graphs. In consequence, existing solutions still suffer from much unnecessary I/O traffic, because they have to entirely load each partition into the memory for concurrent jobs even if most vertices in this partition are inactive and may be shared by a few jobs. In this paper, we propose an efficient structure-aware storage system, called GraphSO, for higher throughput of the execution of concurrent graph processing jobs. It can be integrated into existing out-of-core graph processing systems to promote the execution efficiency of concurrent jobs with lower I/O overhead. The key design of GraphSO is a fine-grained storage management scheme. Specifically, it logically divides the partitions of existing graph processing systems into a series of small same-sized chunks. At runtime, these small chunks with active vertices are judiciously loaded by GraphSO to construct new logical partitions (i.e., each logical partition is a subset of active chunks) for existing graph processing systems to handle, where the most-frequently-used chunks are preferentially loaded to construct the logical partitions and the other ones are delayed to wait to be required by more jobs. In this way, it can effectively spare the cost of loading the graph data associated with the inactive vertices with low repartitioning overhead and can also enable the loaded graph data to be fully shared by concurrent jobs. Moreover, GraphSO also designs a buffering strategy to efficiently cache the most-frequently-used chunks in the main memory to further minimize the I/O traffic by avoiding repeated load of them. Experimental results show that GraphSO improves the throughput of GridGraph, GraphChi, X-Stream, DynamicShards, LUMOS, Graphene, and Wonderland by 1.4-3.5 times, 2.1-4.3 times, 1.9-4.1 times, 1.9-2.9 times, 1.5-3.1 times, 1.3-1.5 times, and 1.3-2.7 times after integrating with them, respectively.

Index Terms—Iterative graph processing, out-of-core, concurrent jobs, storage system

1 Introduction

With the huge demand for analyzing graph data, massive concurrent iterative graph processing jobs are often executed on the common platforms (e.g., Giraph [1], Pregel [2], GraphJet [3], and Plato [4], [5]) of many enterprises (e.g., Facebook [6], Google [7], Twitter [8], and Tencent [9]) daily to exploit various properties of their underlying graphs offline, for different applications (e.g., the variants of PageRank [10] for content recommendation, WCC [11] for social media monitoring, and k-means [12] for customer segmentation). Fig. 1 shows the number of jobs that are running concurrently on a common graph processing platform of a real Chinese social network. There can be up to 30 concurrent graph processing jobs. Although many out-of-core graph processing systems, e.g., X-stream [13], GridGraph [15], and DynamicShards [16], are proposed to efficiently handle a single job with high cost efficiency, they surfer from much redundant I/O cost since the same graph data are repeatedly loaded to serve concurrent jobs. Thus, several recent studies [18], [19], [20], [21], [22] propose to eliminate the redundant I/O traffic by enabling concurrent jobs to share the storage and accesses of the same graph. These solutions first divide the graphs into static partitions to reside in the disk, where each partition consists of a number of vertices and their associated edges. During execution, the active static partition (i.e., its vertices are active for some jobs) is loaded into the memory in some special ways to serve concurrent jobs for less data access cost.

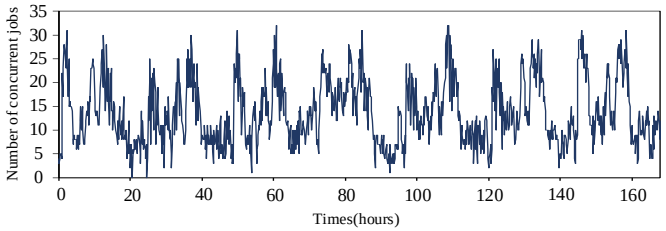


Fig. 1. Number of jobs traced on a social network

However, with these solutions [18], [19], [20], [21], [22], each active partition (i.e., some vertices and edges in this partition need to be processed by a job) needs to be loaded into the memory, even when the most proportion of the

- *Xiaofei Liao, Jin Zhao, Yu Zhang (Corresponding author), Hai Jin, and Lin Gu are with National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.*
  E-mail: {xfliao, zjin, zhyu, hjin, lingu}@hust.edu.cn
- *Bingsheng He is with the Department of Computer Science, National University of Singapore, Singapore 117418.*
  E-mail: hebs@comp.nus.edu.sg
- *Ligang He is with the Department of Computer Science, University of Warwick, Coventry CV4 7AL, United Kingdom.*
  E-mail: liganghe@dcs.warwick.ac.uk

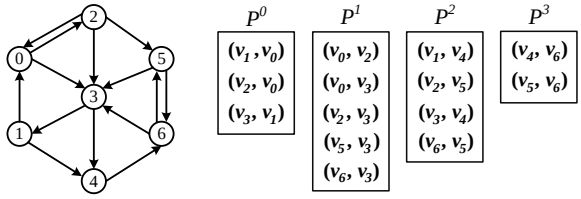
vertices and edges of this partition are not needed by any job or are only needed by a few jobs. In fact, the set of active vertices are usually different for various concurrent jobs and also evolve over time because of their various parameters and different computational complexity. Besides, a few vertices (or called *hub-vertices*) require more iterations to converge than other vertices because of the power-law property [23] of the real-world graphs, which causes a large number of vertices in each active static partition to be inactive for most concurrent jobs at most iterations. Repeatedly loading the graph data associated with these inactive vertices leads to severe I/O inefficiency and also causes the loaded graph data to be shared by only a few concurrent jobs. It eventually incurs low throughput of concurrent jobs.

To tackle this challenge, in this paper, we propose a structure-aware storage system *GraphSO* [1] for concurrent graph processing jobs. Through the provided APIs, GraphSO can be integrated into existing out-of-core graph processing systems to efficiently handle concurrent iterative graph processing jobs, without modifying the applications on these systems. Specifically, GraphSO logically divides the static partitions into a series of small same-sized chunks for fine-grained management of the graph storage. At runtime, only the chunks with active vertices are judiciously loaded by GraphSO to construct new logical partitions with the suitable size required by existing graph processing system. In this way, it can efficiently reduce the unnecessary I/O overhead resulted from loading the graph data associated with the inactive vertices of the active static partitions. Because some chunks (e.g., the ones required by more jobs or with more hub-vertices) may need to be repeatedly processed by concurrent jobs at many iterations, these chunks are preferentially loaded to construct the logical partitions to serve the execution of the concurrent jobs. The other ones are delayed so as to make them wait to be required by more jobs. It enables the generated logical partitions to be reused by as many jobs as possible. After that, each generated logical partition is handled by concurrent jobs in a novel synchronous way. Only when a chunk of the generated logical partition has been handled by the related jobs, the next chunk can be processed. When all chunks of a logical partition have been handled, the next logical partition can be generated for processing. By doing so, it can regularly stream the chunks into the main memory and the *Last-Level Cache* (LLC) to maximize the chunk reuse in the main memory and the LLC, to lower the data access cost. Finally, GraphSO tries to cache the frequently-used chunks in the main memory via a structure-aware buffering strategy, which can further minimize the I/O traffic by eliminating the repeated I/Os for loading these chunks.

This paper mainly owns the following contributions:

- We disclose the significant I/O inefficiencies of existing out-of-core concurrent graph processing solutions and observe that it is the main problem that slows down overall performance.
- We propose a structure-aware storage system that enables existing out-of-core graph processing systems to efficiently and effectively handle concurrent jobs by reducing the unnecessary I/O traffic and maximizing the utilization of the loaded graph data.
- We design a structure-aware buffering strategy to further minimize I/O overhead.
- We integrate GraphSO into seven popular out-of-core graph processing systems, i.e., GridGraph [15], GraphChi [14], X-Stream [13], DynamicShards [16], LUMOS [17], Graphene [24], and Wonderland [25]. Extensive results show that GraphSO can improve the throughput of concurrent jobs by 1.4-3.5 times, 2.1-4.3 times, 1.9-4.1 times, 1.9-2.9 times, 1.5-3.1 times, 1.3-1.5 times, and 1.3-2.7 times for GridGraph, GraphChi, X-Stream, DynamicShards, LUMOS, Graphene, and Wonderland, respectively.

1. The source code of GraphSO is available at: https://github.com/Program-Anonymity/GraphSO.git



(a) Example graph (b) Edge partitions

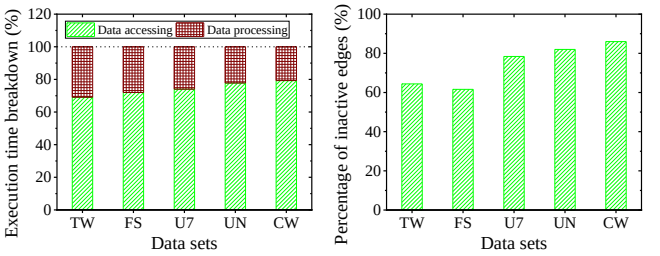
Fig. 2. Example to divide a graph into four static partitions (P^0, P^1, P^2, and P^3) that reside on disk

The remainder is organized as below. The I/O inefficiencies of existing solutions are discussed in Section 2. Section 3 presents the system design and the implementation of GraphSO. Section 4 describes the experimental evaluation. The related works are reviewed in Section 5. Finally, this work is concluded in Section 6.

2 Background and Motivation

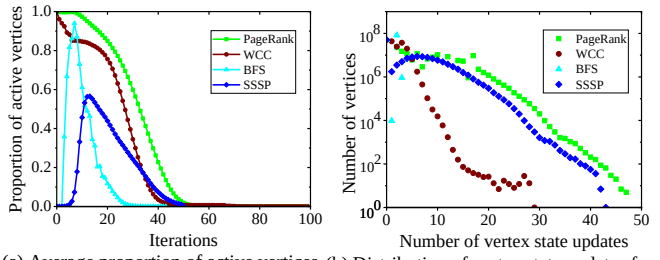
2.1 Out-of-Core Concurrent Graph Processing

Recently, several solutions [19], [20] are proposed to enable multiple iterative graph algorithms to concurrently analyze the same large real-world graphs for high performance efficiency on a single machine. The graph data for each iterative graph processing job consist of the job-specific vertex value data (i.e, S), e.g., the distances from source vertex for SSSP [26] and the ranking scores for PageRank [10], and the graph structure data (i.e., G). To get better performance, existing solutions enable the storage and data accesses of G to be shared by concurrent jobs because G is the same for these jobs. Specially, the graph structure data are first partitioned into static partitions, which are usually first stored on the disk. Fig. 2 depicts an example graph partitioned into four static partitions, and each of which is comprised of a subset of vertices along with these vertices' incoming edges. Within each iteration, the active static partitions for each job are first identified, and the intersection of these partitions for different jobs is then derived. After that, the shared static partitions can be gotten for concurrent jobs and are sequentially loaded to serve these jobs. To exploit the sequential performance of disk, existing solutions usually entirely load each static partition into the memory. After processing all of the active static partitions, the vertex value data of these jobs will be updated and the active static partitions that need to be processed at the next iteration of each job can be obtained. Note that the static partitions will be repeatedly traversed to serve concurrent jobs until all static partitions become inactive for these jobs.



(a) Breakdown of total execution time into graph data accessing and processing times (b) Percentage of the number of inactive edges to that of all loaded edges

Fig. 3. The execution statistics of concurrent jobs: (a) Breakdown the total execution time into graph accessing and graph processing times, (b) Percentage of inactive edges to total loaded edges during the execution.



(a) Average proportion of active vertices in each static partition on CW dataset (b) Distribution of vertex state updates for various graph algorithms on CW dataset

Fig. 4. (a) average proportion of the number of active vertices to that of vertices in each active static partition for jobs over CW dataset, (b) distribution of the number of vertex state updates for jobs in CW dataset, where the vertices are sorted by the number of their updates.

2.2 Inefficiency of I/O Performance

While the above solutions leverage disk locality, it suffers from much unnecessary overhead. During the execution, the static partitions may contain both active and inactive vertices. Entirely loading each static partition within every iteration incurs the useless effort for loading and processing the graph data associated with inactive vertices. This wasteful effort is the main reason to slow down the overall performance because (1) the concurrent graph processing cost is dominated by graph data accessing, and (2) at the most iterations, a large proportion of vertices in the static partitions are inactive for concurrent jobs. Fig. 3 (a) depicts the execution time breakdown for concurrent jobs on the cutting-edge out-of-core concurrent graph processing system called GridGraph-M (i.e., GridGraph [15] integrated with GraphM [20]) over five real-world graphs. The benchmarks and platform will be introduced in detail in Section 4.1. From Fig. 3 (a), we can find that the graph data accessing time occupies a large ratio of total execution time, and the ratio increases as the graph size becomes larger. To understand how much unnecessary I/O overhead is incurred, we evaluate the percentages of the loaded inactive edges (i.e., the loaded edges with inactive source vertices) to the total loaded edges for concurrent jobs on GridGraph-M. As depicted in Fig. 3 (b), there indeed exists significantly unnecessary I/O overhead. For example, over *CW*, GridGraph-M takes over 86% I/O traffic to load the inactive edges for concurrent jobs.

We attribute these phenomena to the real-world graphs' power-law property [23]. Specifically, a small part of vertices (called hub-vertices) are neighbored with most vertices. Generally, hub-vertices require more iterations to converge than others, because their values are impacted by most vertices. Thus, only a small part of vertices in the static partitions are active for concurrent jobs after a few iterations. To prove this phenomena, we evaluated the details of the data accesses characteristic of concurrent jobs on GridGraph-M over CW dataset. Fig. 4(a) depicts the average proportion of the number of active vertices to that of vertices in each partition across iterations in different algorithms. We can observe that the ratio of active vertices drops significantly after a few iterations and becomes very low as the algorithms close to convergence. Besides, the active vertices are different for various algorithms in each iteration and evolve with time. Fig. 4(b) depicts the distribution of the number of vertex state updates when performing different algorithms. It shows that a few vertices are frequently handled by concurrent jobs than others, because these vertices connect with most vertices. Thus, the most ratio of the I/O traffic is carried out for loading the graph data associated with these vertices. It motivates us to propose a structure-aware graph storage optimizations based on the set of active vertices of the concurrent jobs to solve the I/O inefficiency of concurrent jobs for higher throughput.

3 Design of GraphSO

The structure-aware storage system proposed in this paper is named GraphSO. By using several APIs provided by GraphSO, it can be integrated into existing out-of-core graph processing systems to efficiently manage the storage and data access of the same graph for concurrent jobs. First, it logically divides the static partitions into fine-grained chunks so as to provide an opportunity to transparently minimize the unnecessary I/O overhead. After that, it judiciously loads the active chunks to repartition them into the logical partitions to serve the execution of concurrent jobs. In this way, the graph data associated with many inactive vertices in the static partitions do not need to be loaded along with the active ones in these static partitions, and the generated logical partitions can be fully shared by concurrent jobs. Finally, the frequently-used chunks are buffered in the memory to further reduce I/O traffic.

3.1 Overview of GraphSO

Fig. 5 shows the architecture of GraphSO, and its key techniques mainly include structure-aware adaptive graph repartitioning/processing and structure-aware buffering.

Structure-aware Adaptive Graph Repartitioning and Processing. Before the execution, the graph data usually needs to be preprocessed. The partitioning methods and graph formats for different graph processing systems are usually varied. Therefore, for each graph processing system integrated with GraphSO, the original graph stored in GraphSO is first divided into static partitions according to the partitioning methods of this system, and then the static partitions are respectively converted into the graph format adopted in the graph processing system (e.g., the grid format for GridGraph [15] and the shard format for GraphChi [14]). We do not modify the graph format used by any graph processing systems for transparent integration of GraphSO. The static partitions are then stored in the disk. To achieve fine-grained graph management (i.e., graph repartitioning and buffering) as the following described, the static partitions are logically divided into small same-sized chunks, and the information of the small chunks are stored in a *TChunk* table. Each chunk should be fit in the LLC to exploit the cache locality.

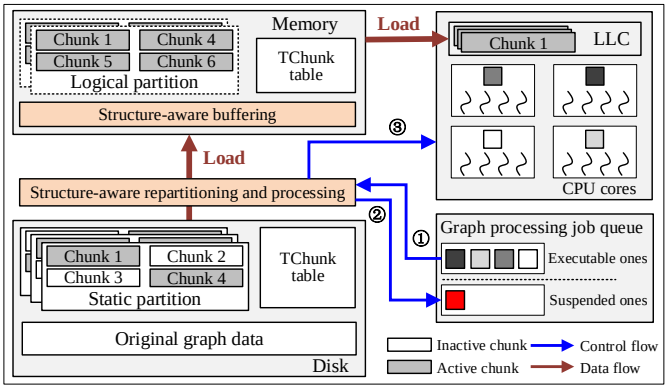


Fig. 5. The architecture of GraphSO

After that, the graph data needs to be loaded into the memory to serve the execution of concurrent jobs. To reduce the unnecessary I/O for loading the unnecessary graph data, it identifies the active chunks (i.e., C_{active}) before each iteration. Then, GraphSO logically repartitions the graph by loading and assigning the active chunks together to construct the logical partitions to serve concurrent jobs. Note that the important chunks (e.g., the ones are required by more jobs or own many hub-vertices) are distinguished and preferentially loaded by GraphSO to generate the logical partitions, enabling the generated logical partitions can be shared by more jobs. The other chunks are delayed to wait to be required by more jobs. This procedure is drafted as an API as: $LP_j^i \leftarrow Schedule(C_{active}, |P|, Load())$, where $|P|$ is the size of the static partitions required by the related graph processing system, LP_j^i is the generated logical partition in the memory and is loaded by job j. LP_j^i can be shared by its related jobs (the ones need to handle it), and the unrelated jobs are suspended until their active partitions are loaded. Note that the next logical partition can be generated to be handled by concurrent jobs only if the generated logical partition has been processed by all of its related jobs.

Besides, the workload of each logical partition is usually skewed between various jobs because of their different active vertex number and computational complexity. Thus, when processing each generated partition, it unevenly allocates the computing resources to concurrent jobs so as to synchronously handle each chunk in the LLC. In this way, each chunk can be reused by different jobs when it is loaded into the LLC, in order to lower the data access cost. Note that, although the important chunks may be loaded for the concurrent jobs multiple times, each job handles the loaded chunks only if this job needs to process this loaded chunks at this moment. Therefore, GraphSO does not affect the correctness of the final results of each job.

Structure-aware Buffering. To further reduce the data access cost and fully utilize the memory resources, several chunks are selected and cached in the main memory. In detail, the important chunks need to be preferentially cached in the memory because of their higher loading frequency during the execution. Thus, a buffering strategy is designed to preferentially cache the important chunks for the concurrent jobs. In this way, the repetitive I/O traffic for loading the cached chunks can be reduced for concurrent jobs.

Programming APIs. To transparently use GraphSO, several APIs need to be inserted into existing graph processing systems, while the graph applications executed on these systems do not need to be modified. Specifically, $Init()$ is adopted to initialize GraphSO through dividing the static partitions into the fine-grained chunks as discussed in Section 3.2.1. $GetActiveChunks()$ and $Schedule()$ are inserted between successive iterations in the program to get the active chunks and efficiently load the generated logical partitions for the concurrent jobs, respectively. Note that, $GetActiveChunks()$ needs to be placed before $Schedule()$ to identify the active chunks for graph repartitioning.

3.2 Repartition-centric Execution of Concurrent Jobs

In most iterations, there may be only a few vertices that are active for concurrent jobs due to the power-law property [23], incurring much unnecessary I/O traffic for loading the graph data associated with the inactive vertices in each static partition. Although some solutions [16], [27] are proposed to dynamically repartition the graph, the set of active vertices is usually different for various jobs and the number of the related jobs of different active vertices is usually skewed. Therefore, when using existing solutions [16], [27] to support the execution of the concurrent jobs, the graph will be repartitioned individually and gets different repartitioned results for different jobs. It incurs high repartitioning cost. Furthermore, the layouts of the same underlying graph are different for different jobs, and thus multiple copies of the same graph have to be accessed and maintained for these jobs individually. In addition, when integrating existing repartitioning solutions [16], [27] into the existing concurrent graph processing systems (e.g., GraphM [20]) to serve multiple concurrent jobs, the frequently accessed active vertices (i.e., the ones to be processed by much more concurrent jobs) may be repartitioned together with the other infrequently accessed active vertices (i.e., the ones to be processed by only a few concurrent jobs or even only one job). It causes low utilization of the loaded dynamic partitions and also high repartitioning overhead.

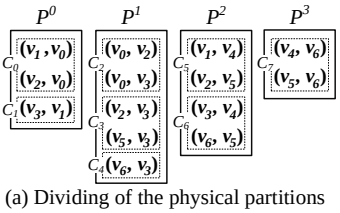
To address the above problems, we propose a structure-aware adaptive graph repartitioning and processing scheme to efficiently serve the execution of concurrent jobs. Specifically, it first divides the static partitions of existing graph processing systems into fine-grained chunks so as to provide an opportunity to effectively reduce the unnecessary I/O traffic efficiently. After that, at the execution time, the set of active chunks are dynamically identified and adaptively repartitioned into the common logical partitions for different concurrent jobs, and then trigger these jobs to synchronously handle these logical partitions. Thereby, it enables these jobs to effectively share the repartitioned results and the accesses to the same graph data. Note that, the most-frequently-used chunks are tried to be assigned into the same logical partitions together, which enables the logical partitions to be fully shared by the concurrent jobs. It enables higher utilization of the loaded graph data and also much lower repartitioning cost.

3.2.1 Fine-grained Graph Representation

To achieve fine-grained graph management, the static graph partitions need to be logically divided into small chunks. When using the existing fine-grained partitioning strategy [14], [15], [24], it may cause much redundant data access cost or high synchronization overhead for processing each chunk. Specifically, if the chunk size is larger than the LLC

Algorithm 1 Pseudo Code of Fine-grained Dividing

```
1: procedure DIVIDE(P^i, TChunk^i, |C|)
2:     Initial(TChunk^i)
3:     c ← 0  /*c indicates the c^th chunk in P^i*/
4:     for each edge e ∈ P^i do
5:         if e_s ∈ TChunk^i_c then
6:             TChunk^i_c.N^+(e_s) ← TChunk^i_c.N^+(e_s) + 1
7:         else
8:             TChunk^i_c.Insert(⟨e_s, 1⟩)
9:         end if
10:        TChunk^i_c.N_e ← TChunk^i_c.N_e + 1
11:        if TChunk^i_c.N_e ≥ |C|/|E| or P^i is traversed then
12:            if c equals zero then
13:                TChunk^i_c.Loc ← Location(P^i)
14:            else
15:                TChunk^i_c.Loc ← TChunk^i_{c-1}.Loc + |C|
16:            end if
17:            c ← c + 1
18:        end if
19:    end for
20: end procedure
```



(a) Dividing of the physical partitions

Chunk ID	Location	Key-value pairs
C_0	Loc_0	<v_1, 1>, <v_2, 1>
C_1	Loc_1	<v_3, 1>
C_2	Loc_2	<v_0, 2>
C_3	Loc_3	<v_2, 1>, <v_5, 1>
C_4	Loc_4	<v_6, 1>
C_5	Loc_5	<v_1, 1>, <v_2, 1>
C_6	Loc_6	<v_3, 1>, <v_6, 1>
C_7	Loc_7	<v_4, 1>, <v_5, 1>

(b) *TChunk* table

Fig. 6. Illustration of the fine-grained graph dividing, where the example graph partitions are divided into eight chunks and the suitable size of each chunk is assumed to be the total size of two edges

size, it causes much redundant data access cost, because of cache thrashing and intense contention among concurrent jobs. If the chunk size is too small, it may cause frequent synchronization among the concurrent jobs, because the next chunk is allowed to be loaded by GraphSO only if the concurrent jobs have processed the current chunk. Therefore, for better cache locality and higher system performance, the suitable chunk size (i.e., $|C|$) should fit into that of the LLC. Thus, the value of $|C|$ is determined by:

$$|C| \times N + \frac{|C| \times N}{|G|} \times N_v \times |V| + s \leq |LLC|, \qquad (1)$$

where $|V|$ denotes the size of vertex value, $|G|$ indicates the size of graph structure data. N and N_v represent the number of CPU cores and graph's vertices, respectively. $|LLC|$ is the LLC's size, s is the reserved space size. Because each chunk is usually handled by one CPU core, the number of chunks loaded in the LLC equals that of CPU cores (i.e., $|C| \times N$). Note that $|C|$ is set as the largest integer that satisfies (1). With such settings, the same chunk can be reused by all related jobs once it is loaded into the LLC, while incurring low synchronization cost at the same time.

The static partitions are logically divided into a series of chunks by traversing the graph once, as shown in Fig. 6(a). During the traverse, a *TChunk* table is established to record the information of chunks. Each entry of the *TChunk* table is corresponded to a visited chunk and consists of three fields. As Fig. 6(b) depicts, the first two fields describe the ID and the start location (denoted by Loc) of the corresponding chunk in the graph file. The third field is a key-value table to store the source vertices set and the number of their outgoing edges in this chunk. For each key-value pair, the source vertices IDs (e.g., v_i) in this chunk are the keys and the number of the vertices' outgoing edges (e.g., $N^+(v_i)$) of this chunk are the values.

The procedure of fine-grained dividing of the static partitions, e.g., P^i, is shown in Algorithm 1. The $TChunk^i$ is first initialized to store the information of the chunks in P^i (Line 2). Then, the edges in P^i are sequentially traversed to collect the edges' source vertices (i.e., e_s) and the number of these vertices' outgoing edges (i.e., $N^+(e_s)$) in each chunk (Lines 5-9). $TChunk^i_c.N_e$ is used to count the number of edges in chunk c. When $TChunk^i_c.N_e$ enables the size of chunk c to be the suitable chunk size or all edges in P^i has been traversed (Line 11), the start location of chunk c (i.e., $TChunk^i_c.Loc$) can be obtained (Lines 12-16). Note that the start location of P^i is set as $TChunk^i_c.Loc$ if chunk c is the first chunk in P^i. Then, it moves to store the the next chunk's information (Line 17). Note that, when the graph is evolved, the consistent snapshots of the concurrent jobs can be maintained by the solution employed in GraphM [20] and the $TChunk$ also needs to be updated.

3.2.2 Adaptive Graph Repartitioning for Concurrent Jobs

During the execution, GraphSO dynamically identifies the active chunks for different concurrent jobs. Based on these collected information, GraphSO adaptively repartitions the underlying graph at runtime by judiciously putting the active chunks of all concurrent jobs together into several logical partitions. These generated logical partitions then are provided to be synchronously handled by all concurrent jobs and can also be shared by them. By such means, the graph repartitioning cost is amortized by these concurrent jobs and these jobs only need to maintain a single copy of the same underlying graph on the memory and LLC.

In practice, a chunk will be loaded into the memory for processing only if this chunk is active for some jobs (i.e., its vertices are active for these jobs). Note that, if the vertices of a chunk are updated by some jobs in the current iteration, this chunk can be identified as active chunk for these jobs in the next iteration. Thus, we can get the active chunks in the next iteration by tracing the vertex update in the current iteration [28]. In this way, the I/O overhead of loading the inactive chunks in active static partitions can be eliminated.

Specifically, only the active chunks are expected to be loaded by GraphSO to construct the logical partitions, where the size of each logical graph partition needs to meet the requirement of the integrated graph processing system so as to make GraphSO transparent to the existing graph processing systems. It is because existing out-of-core graph processing systems [13], [14], [15], [17] usually load and process the graph data by taking the partition as the basic unit, where the suitable size of the partition (i.e., the static partition size) is determined by the corresponding graph processing system and is usually different for various cases (e.g., X-Stream [13] usually uses different partition sizes for different graph for better performance). However, unsuitable loading order of the chunks may incur underutilization of the loaded graph data. First, for different chunks, the number of the related jobs and the number of the active vertices are usually

Algorithm 2 Pseudo Code of Repartition-centric Execution

```
1: procedure SCHEDULE(C_active, |P|, Load())
2:     LP^i ← GenerateLogicalPartition(C_active, |P|)
3:     J^i ← GetRelatedJobs(LP^i)
4:     ResumeJob(J^i)
5:     if j∉J^i then
6:         SuspendJob(j)
7:     end if
8:     RemoveJob(j, J^i)
9:     return LP^i
10: end procedure
11: procedure GENERATELOGICALPARTITION(C_active, |P|)
12:     /*Get the top |P|/|C| chunks based on their priorities*/
        LP^i ← GetChunks_Pri(C_active)
13:     Sort_ID(LP^i)
14:     if LP^i has not be loaded into the memory then
15:         Buf ← SetSharedBuffer(LP^i)
16:         for each chunk C_m∈LP^i do
17:             Buf ← Load(TChunk_m.Loc, TChunk_m.N_e)
18:         end for
19:     else
20:         Buf ← ObtainFromSharedBuffer(LP^i)
21:     end if
22:     return Buf
23: end procedure
```

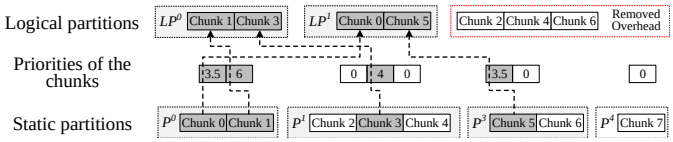


Fig. 7. Illustration of adaptive graph repartitioning, where *chunk 0* and *chunk 1* of P^0, *chunk 3* of P^1, and *chunk 5* of P^2 are active for concurrent jobs and the remaining chunks are inactive

skewed and are also dynamically changed over time. The chunks associated with more related jobs or with more active vertices may be shared by more concurrent jobs and have a greater impact on vertex state propagation. Second, the hub-vertices are connected with most vertices of the graph, and thus the states of most vertices are affected by that of these hub-vertices. The delayed processing of the chunks containing these hub-vertices may cause many vertices fail to be activated for the concurrent jobs. Therefore, the chunks may only be shared by a few jobs or even only one job, when they are arbitrarily loaded into the memory to construct the logical partitions for the processing of the concurrent jobs. It results in poor performance.

Because processing the graph data along various orders does not affect the accuracy of vertex values [25], [29], [30], the active chunks are judiciously loaded by GraphSO to construct the logical partitions, thereby enabling the loaded graph data to be shared by more jobs. The key idea is to preferentially load the most-frequently-used chunks, i.e., the ones required by more jobs or owning more hub-vertices, to construct the logical partitions. In this way, more chunks can be activated for concurrent jobs and most chunks in each loaded logical partition can be repeatedly accessed by more concurrent jobs, thus enabling higher utilization of each loaded logical partition. To achieve this goal, a priority is assigned to each active chunk to schedule their loading order. In essential, it is set by two rules. First, the chunk is assigned with the higher priority when it has more related jobs and more active vertices. Second, the vertices of the chunk with the higher average degree need to be assigned with higher priorities. Thus, the priority, e.g., $Pri(C_m)$, of each active chunk, e.g., C_m, can be determined by:

$$Pri(C_m) = N(J_m) \cdot N(A_m) \cdot \frac{\sum_{v_i \in V_m} D(v_i)}{N(V_m)}, \qquad (2)$$

where V_m is the set of source vertices in C_m, $D_i(v_i)$ denotes the degree of vertex v_i, and $N(V_m)$ represents the number of source vertices in C_m. $N(J_m)$ is the number of related jobs for C_m. $N(A_m)$ is the number of active vertices in C_m. $N(V_m)$, $D(v_i)$ and the initial values of $N(A_m)$ and $N(J_m)$ can be obtained at preprocessing phase, while $N(A_m)$ and $N(J_m)$ are incrementally updated during the execution. Then, the loading order of the chunks can be scheduled according to their priorities.

After that, the active chunks are loaded and reassigned together by GraphSO to construct new logical partitions according to the procedure *GenerateLogicalPartition*() in Algorithm 2, where the size of each logical partition needs to meet the requirement of the integrated graph processing system because the suitable size of partition (i.e., the static partition size $|P|$) is different for various systems. Specifically, it first gets the suitable number of the active chunks based on their priorities to construct a logical partition LP^i (Line 12). Then, these chunks of LP^i are sorted according to their chunk IDs, thereby providing an opportunity to exploit the sequential disk I/O (Line 13). If LP^i has not loaded into the memory, a shared buffer (i.e., *Buf*) is set up to store it (Lines 14-15), and the chunks of LP^i are sequentially loaded into *Buf* (Lines 16-18). Elsewise, LP^i can be obtained in the shared memory (Line 20). Finally, the LP^i stored in *Buf* is used as the returned results to be handled by the concurrent jobs (Line 22). Fig. 7 shows how to repartition the graph. We can observe that the I/O traffics associated with the chunks $chunk2$, $chunk4$, and $chunk6$ can be removed by our repartitioning strategy.

3.2.3 Efficient Processing of Logical Partitions

When a logical partition is generated and loaded, its related jobs are triggered to handle it, while the unrelated jobs will be suspended to wait for loading their active logical partitions. When the current logical partition has been processed by its related job, the next logical partition will be generated. This procedure $Schedule()$ is described in Algorithm 2. In detail, when $Schedule()$ is called by a job j, it first generates and loads a logical partition LP^i by calling the procedure *GenerateLogicalPartition*(), and then the related jobs (i.e., J^i) of LP^i are obtained (Line 3). The related jobs in *suspended_queue* are resumed to handle LP^i (Line 4). If LP^i does not need to be handled by the current job j, the job j will be suspended (Lines 5-7). Then, it removes job j from J^i (Line 8). Finally, the logical partition LP^i is provided as a return result to job j for processing (Line 9).

To efficiently stream the logical partitions into the LLC, the processing of their chunks need to be synchronized for different jobs. Only when a streamed chunk has been handled by its related jobs, the next chunk can be handled. However, the computational load of a chunk is usually skewed for different jobs because of the heterogeneity in

TABLE 1
Proprieties of Graph Datasets

Datasets	Vertices	Edges	Sizes
Twitter(TW) [33]	41,652,230	1,468,365,182	17.5 GB
Friendster(FS) [34]	65,608,366	1,806,067,135	22.7 GB
Uk-2007(U7) [33]	105,896,555	3,738,733,648	46.2 GB
Uk-union(UN) [33]	133,633,040	5,507,679,822	68.3 GB
Clueweb12(CW) [33]	978,408,098	42,574,107,469	317 GB

both the computational complexity and the number of active vertices in current chunk. The computational load of job j for the processing of chunk C_m can be gotten by $L_j(C_m) = \theta_j \cdot \sum_{v_i \in V_m \cap A_j} N_m^+(v_i)$, where A_j is the active vertices set of the job j in current iteration, $N_m^+(v_i)$ denotes the number of v_i's outgoing edges in C_m, and θ_j is the profiled average execution time to process an edge in job j. After that, it unevenly assigns the computing resources to different jobs according to their calculated computational load. In this way, each chunk can be reused in the LLC by its related jobs, thereby reducing the data access cost.

3.3 Structure-aware Buffering for Concurrent Jobs

In practice, the same graph data are usually repeatedly accessed by multiple concurrent jobs, because these jobs need to iteratively handle this graph iteration by iteration until convergence. Besides, as shown in Fig. 4(b), some graph data need to be accessed more frequently by these jobs. Therefore, there is an opportunity to improve the performance of the concurrent jobs by effectively buffering the frequently-used graph data in the main memory. Although many graph processing systems [25], [28], [31], [32] have used the caching technique to reduce the I/O overhead, they still suffer from much redundant data storage and access cost when handling concurrent jobs. Specifically, the existing solutions [25], [28], [31], [32] are designed to buffer the graph data in the main memory for each job separately, and multiple copies of the same graph data and also many graph data which are not important to multiple concurrent jobs will be buffered in the main memory when supporting the execution of multiple concurrent jobs. These graph data cause serious contention for the main memory, which leads to significant data thrashing. Besides, the number of the related jobs associated with different chunks is usually skewed and varies over time. When using the existing solutions [25], [28], [31], [32] to support the execution of multiple concurrent jobs, the buffered graph data may be needed by only a few jobs (or even one job). It eventually results in the low utilization of the buffered graph data.

To address this challenge, we propose an efficient buffering strategy to judiciously buffer only one copy of the common chunks in a buffer (called buffer space) of the main memory to be shared by concurrent jobs, thereby enabling the buffered graph data to be reused by more jobs to further reduce data access cost. Specifically, the chunks with more related jobs and more active vertices need to be buffered in the buffer space because most proportion of the graph data in these chunks can be reused by more concurrent jobs. Besides, the chunks with more hub-vertices need to be buffered in the buffer space because they need to be repeatedly loaded into the main memory. Therefore, the most-frequently-used chunks (i.e., the ones with higher priority determined by the Formula (2)) are tried to be preferentially cached in the buffer space. In this way, the I/O traffic for loading these chunks can be efficiently spared by shielding the frequently-used graph data from thrashing and enabling the cached chunks to be effectively reused by more concurrent jobs, thereby minimizing the I/O cost.

To achieve this goal, we customize an insertion policy and an eviction policy to efficiently cache the chunks. Within each iteration, the active chunks are loaded into the memory and cached by the following strategies:

- *Insertion Policy*: When the buffer space is not full and the loaded chunks have not cached, these chunks need to be inserted into buffer space.
- *Eviction Policy*: If the buffer space is full and the loaded chunk have not been cached yet, the cached chunk, e.g., C_m, with the lowest priority, e.g., $Pri(C_m)$, will be evicted.

By this way, the memory resources are fully utilized and the I/O traffic is further reduced by reusing the cached chunks in the main memory.

4 Evaluation

4.1 Experiment Setup

The hardware platform adopted in the experiments is a single machine, which contains two Intel Xeon E5-2670 CPUs (each CPU owns 8 cores and 20 MB LLC), a 32 GB main memory and a standard 1 TB hard drive, running Linux kernel 2.6.32. All programs are compiled with gcc version 4.9.4 and cmake version 3.11.0. The datasets used in our experiments are five real-world graphs as detailed in Table 1, where their edge weights are generated randomly from the range [0, 256]. The benchmarks employ four popular graph processing algorithms, which containing: PageRank [10], Breadth-First Search (BFS) [35], Single Source Shortest Path (SSSP) [26], and Weakly Connected Component (WCC) [11]. These algorithms can be classified into two categories: *all-active algorithm* and *non-all-active algorithm*. Specifically, the former (e.g., PageRank and WCC) traverses the graph starting from all vertices at the beginning, whereas the latter (e.g., SSSP and BFS) starts from a subset of vertices at the beginning. Note that, in the real trace of Fig. 1, the proportion of the former is about 34.2%, and that of the latter is 65.8%. In our experiments, we simultaneously submit these algorithms to generate a specified number of jobs, where SSSP and BFS are submitted with the randomly selected starting vertices. The ratio of physical memory (i.e., available memory resource) used to cache graph data is set as 25% by default. The chunk size $|C|$ is determined by the Formula (1). For example, $|C| \leq 1.4$ MB for FS with 8 MB reserved space size, thus $|C|$ is set to 1 MB by GraphSO.

To evaluate the performance of GraphSO, we integrate it into seven popular systems, i.e., GridGraph [15], GraphChi [14], X-Stream [13], DynamicShards [16], LUMOS [17], Graphene [24], and Wonderland [25]. In our experiments, we mainly show the performance improvement when GraphSO is integrated into GridGraph because it is the one of the state-of-the-art and the most popular graph processing systems [17], [25]. We tersely discuss the results when integrating with the other systems in Section 4.7. Specifically, GridGraph integrated with GraphSO (called GridGraph-O in our experiments) is also compared

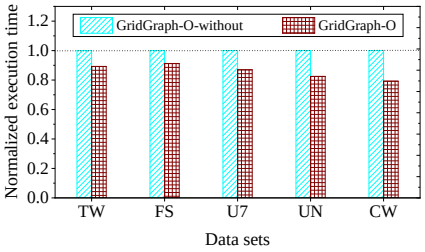


Fig. 8. Execution time for the four jobs without/with our structure-aware buffering strategy

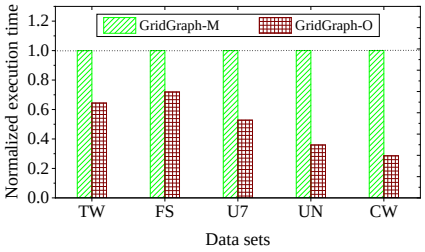


Fig. 9. Comparison of execution time for running the four jobs on various schemes

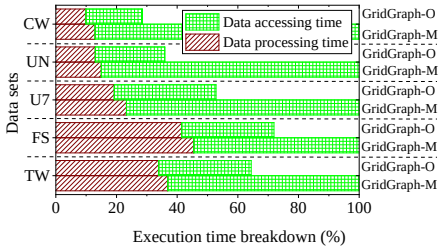


Fig. 10. Execution time breakdown of the four jobs on various schemes

with another scheme, i.e., GridGraph-M [20] (GridGraph integrated with GraphM). GraphM is the state-of-the-art graph storage system that allows the same graph data to be shared by the concurrent jobs on existing graph processing system. Note that the experiments are also conducted in the same way for GraphChi, X-Stream, DynamicShards, LUMOS, Graphene, and Wonderland in Section 4.7. Note that Graphene performs on eight Samsung NVMe 950 Pro 512GB SSDs. Besides, we also evaluate the performance of CGraph [19] to further demonstrate the efficiency of GraphSO. All experiments are conducted for ten times and the reported results are the average value.

We use DynamicShards and LUMOS to illustrate how to integrate GraphSO with existing graph processing systems. For DynamicShards, we insert *GetActiveChunks*() and *Schedule*() between successive iterations in the program to replace its repartitioning logic and the original graph load logic to achieve the lower repartitioning cost and the higher utilization of the loaded graph data. For LUMOS, we use the *GetActiveChunks*() and *Schedule*() to repartition its primary layout partitions and efficiently load the graph structure data for concurrent jobs.

4.2 Preprocessing Cost

Table 2 depicts the preprocessing time of GridGraph, GraphChi, X-Stream, DynamicShards, LUMOS, Graphene, and Wonderland after integrating with GraphM and GraphSO, respectively. We can find that little additional time is required by GridGraph-O, GraphChi-O, X-Stream-O, DynamicShards-O, LUMOS-O, Graphene-O, and Wonderland-O, to generate the *TChunk* compared with GridGraph-M, GraphChi-M, X-Stream-M, DynamicShards-M, LUMOS-M, Graphene-M, and Wonderland-M, because GraphSO needs to collect little more information than GraphM. For example, the additional time of GridGraph-O is about 5%, 7.2%, 6.5%, 4.2%, and 5.7% for TW, FS, U7, UN, and CW, respectively. The extra storage cost required by GridGraph-O is also small and occupies 16.5% (2.89 GB), 8.6% (1.95 GB), 10.2% (4.71 GB), 9.1% (6.22 GB), and 7.8% (24.73 GB) of the total storage overhead of TW, FS, U7, UN, and CW, respectively. Although GraphSO requires little extra overhead, much unnecessary I/O traffic can be reduced by it, as will be shown later.

4.3 Performance of Structure-aware Buffering Strategy

First, we analyze the optimizations designed in GraphSO. All four algorithms (i.e., SSSP, PageRank, BFS, and WCC) are simultaneously submitted to generate four concurrent jobs, and then trace the execution time of concurrent jobs with/without our proposed optimization. Fig. 8 depicts the results. GridGraph-O and GridGraph-O-without are the version of GridGraph-O with and without our structure-aware buffering strategy (Section 3.3). We can find that GridGraph-O obtains better performance than GridGraph-O-without because the I/O overhead of the buffered chunks is eliminated for concurrent jobs. For example, over CW dataset, the execution time of GridGraph-O is only 79.3% of GridGraph-O-without. Note that, for TW, FS, U7, UN, and CW, the execution time of the GridGraph-O-w/o is 213, 298, 962, 1,012, and 9,909 seconds, respectively.

TABLE 2
Preprocessing time (in seconds)

	TW	FS	U7	UN	CW
GridGraph-M	463.7	716.6	1,803.4	2,681.1	22,401.9
GridGraph-O	486.8	768.2	1,920.7	2,793.6	23,674.3
GraphChi-M	975.3	2,579.2	3,755.9	5,715.9	26,866.3
GraphChi-O	1,052.4	2,814.5	3,922.5	6,145.8	28,013.8
X-Stream-M	252.3	462.2	927.8	1,254.2	9,891.3
X-Stream-O	296.3	506.3	1,185.2	1,427.4	11,265.7
DynamicShards-M	1,034.6	2,475.4	3,472.8	5,125.2	27,013.7
DynamicShards-O	1,146.8	2,591.2	3,615.4	5,517.3	28,916.5
LUMOS-M	579.6	945.9	2,406.7	3,780.4	27,930.8
LUMOS-O	602.3	995.7	2,568.4	3,992.7	29,735.1
Graphene-M	8.2	11.3	116.2	242.7	617.5
Graphene-O	9.1	12.8	127.6	258.3	651.9
Wonderland-M	505.4	804.1	2,146.1	3,244.1	27,554.3
Wonderland-O	528.5	855.6	2,263.3	3,356.6	28,826.7

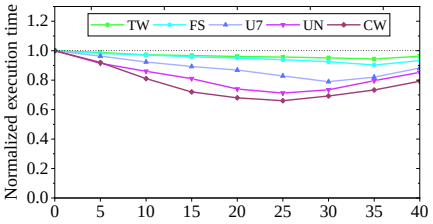


Fig. 20. Execution time of using various ratios of the main memory to buffer the graph data

4.4 Overall Performance

To compare GridGraph-M and GridGraph-O, we concurrently submit SSSP, PageRank, BFS, and WCC as four concurrent jobs to different schemes. The normalized results are depicted in Fig. 9. For TW, FS, U7, UN, and CW, the execution time of the baseline system (i.e., GridGraph-M) is 295, 378, 1,585, 2,071, and 27,493 seconds, respectively. It shows that GridGraph-O requires shorter execution time to run the four jobs than GridGraph-M. This indicates that higher throughput is achieved by GraphSO. For example, comparing with GridGraph-M, GridGraph-O improves the throughput by 3.5 times over CW dataset. The throughput improvement basically results from the lower I/O overhead

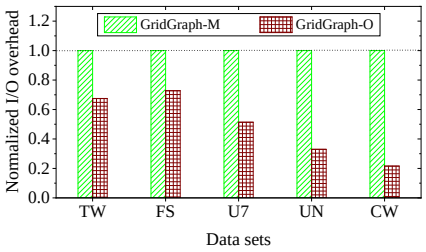


Fig. 11. Normalized I/O traffic for the four jobs on various schemes

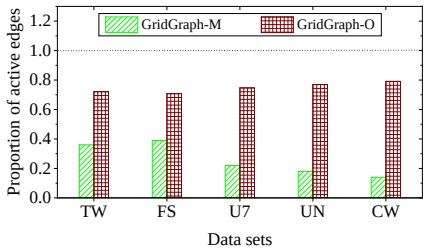


Fig. 12. Proportion of the number of active edges to that of the total loaded edges

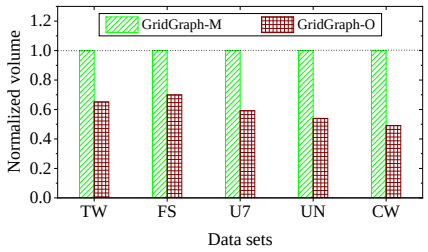


Fig. 13. Volume of the graph data steamed into the LLC for the four jobs with various schemes

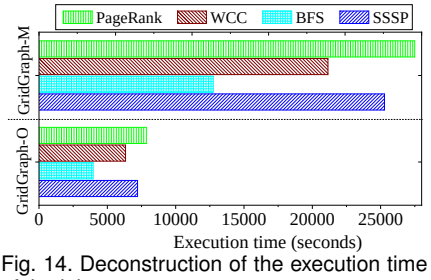


Fig. 14. Deconstruction of the execution time of the jobs

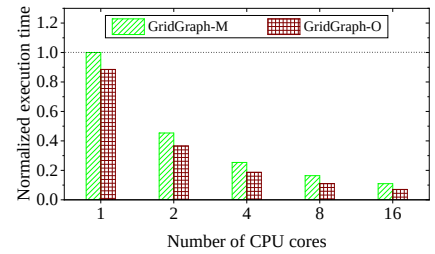


Fig. 15. Total execution time for different number of CPU cores

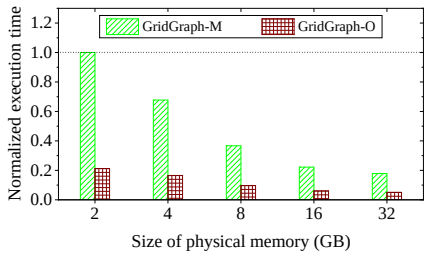


Fig. 16. Execution time for adopting different size of physical memory

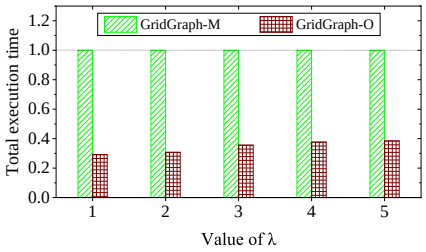


Fig. 17. Total execution time when submitting the jobs with different λ

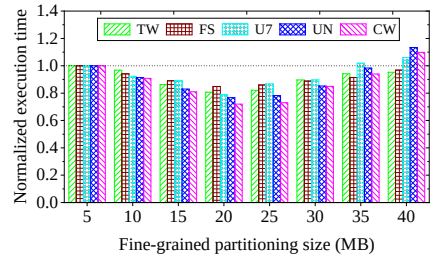


Fig. 18. Total execution time of GridGraph-O with different fine-grained partitioning sizes

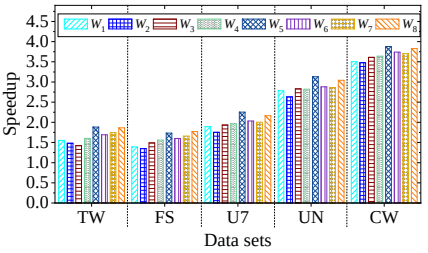


Fig. 19. Performance of GraphSO against GraphM when processing different workloads

brought by GraphSO. Note that the synchronization overhead (i.e., synchronizing multiple jobs to process the same chunks) occupies 6.3%-12.1%, and the chunk management cost (e.g., identifying active chunks and constructing logical partitions) occupies 3.8%-6.7% of the total execution time of the jobs on GridGraph-O for our test instances.

We breakdown the execution time for running the four jobs on various schemes in Fig. 10. It shows that, compared with GridGraph-M, the graph processing time of jobs on GridGraph-O occupies a larger ratio of the total execution time. This means that GridGraph-O requires less data access time. For example, over CW dataset, the data accessing time of GridGraph-O is reduced by 4.6 times compared with GridGraph-M. The lower data access time brought by GraphSO is caused by the following reasons. First, it loads less graph data into the memory, because the loading of much unnecessary graph data (i.e., inactive fine-grained chunks in active coarse-grained partitions) is bypassed and each loaded logical partition (i.e., a set of active chunks) is fully utilized by concurrent jobs. Second, the processing of inactive chunks is skipped in each iteration, resulting in less graph data steamed into the LLC.

In order to prove it, we trace the total I/O traffic for executing the four jobs on various schemes. Fig. 11 shows the normalized results of GridGraph-O against GridGraph-M. As described, compared to GridGraph-M, less I/O overhead is produced by GridGraph-O, especially as the graph size is larger than the physical memory size. For instance, the I/O cost of GridGraph-O is only 21.6% of GridGraph-M over CW dataset, because GraphSO eliminates much unnecessary I/O traffic via repartitioning static partitions and maximizing the utilization of loaded graph data. On the contrary, although GraphM can reduce the redundant I/O overhead by sharing the same graph data for concurrent jobs, much unnecessary graph data are also loaded for concurrent jobs, leading to severe I/O inefficiency.

Fig. 12 depicts the ratio of the number of active edges (i.e., the edges' sources vertices are active) to that of the total edges loaded from disk into the memory with different schemes. We can find that the number of active edges on GridGraph-O occupies the most proportion of the total number of loaded edges, while the proportion is lower in GridGraph-M. This means that the memory resources and the disk bandwidth are efficiently utilized by GraphSO due to the loading and storing of fewer graph data.

Moreover, we trace the volume of graph data streamed into the LLC for running the four jobs on various schemes. The results of GridGraph-O normalized to GridGraph-M are depicted in Fig. 13. The results show that GridGraph-O streams the less data into the LLC in comparison with GridGraph-M. For example, when processing CW dataset, the volume of data streamed into the LLC in GridGraph-O is only 49% of GridGraph-M. This is because GridGraph-M streams the inactive chunks in active static partitions into the LLC for concurrent jobs, while these unnecessary overhead is eliminated by GraphSO.

Fig. 14 evaluated the breakdown of the execution time of the jobs over *CW* dataset. We can observe that each job on GridGraph-O requires a shorter execution time than that on GridGraph-M because the shorter data access time and the

TABLE 3
Workloads of Concurrent Graph Processing Jobs

Workload	Algorithms	Algorithm Heterogeneity
W_1	{PageRank, WCC, SSSP, BFS}	More heterogeneous
W_2	{PageRank, WCC, SSSP, SSSP}	
W_3	{PageRank, WCC, BFS, BFS}	
W_4	{SSSP, SSSP, SSSP, SSSP}	More homogeneous
W_5	{BFS, BFS, BFS, BFS}	
W_6	{SSSP, SSSP, SSSP, BFS}	A mix of similar graph algorithms
W_7	{SSSP, SSSP, BFS, BFS}	
W_8	{SSSP, BFS, BFS, BFS}	

higher CPU utilization are achieved by GraphSO.

4.5 Scalability of GraphSO

To evaluate the scalability of GraphSO, Fig. 15 traces the execution time of the four jobs over various schemes on TW dataset and changing the number of CPU cores. It shows that GridGraph-O outperforms GridGraph-M under any circumstance. In detail, GridGraph-O obtains speedups of 1.13, 1.24, 1.36, 1.49, and 1.55 against GridGraph-M when there are 1, 2, 4, 8, and 16 CPU cores, respectively. This is because GraphSO can efficiently manage the graph structure data to serve concurrent jobs with minimized I/O traffic, while GraphM still suffers from much unnecessary I/O cost.

We then evaluate the performance of various schemes to handle the four jobs over CW dataset when using different size of physical memory. The results are depicted in Fig. 16. We observe that GraphSO achieves better performance improvement when the physical memory is smaller due to its higher utilization of memory resources. Fig. 17 evaluates the performance when the jobs are submitted at different times, and each new job is submitted when the previously submitted job has executed λ number of iterations. It shows that GraphSO outperforms GraphM for all cases because many unnecessary data accesses are eliminated by GraphSO.

Fig. 18 evaluates the performance of GraphSO with different values of fine-grained partitioning sizes, where the size of the LLC of each CPU of our platform is 20 MB. It shows that better performance can be obtained by GraphSO when the value is set as the size of LLC (i.e., 20 MB). When the value is larger than the LLC size, it causes much redundant data access cost due to cache thrashing and intense contention among concurrent jobs. When the value is too small, it causes frequent synchronization among the concurrent jobs, because the next chunk is allowed to be loaded by GraphSO only if the concurrent jobs have processed the current chunk.

Fig. 19 shows the performance of GraphSO against GraphM for different types of the synthesized workloads (listed in Table 3). We can observe that GraphSO outperforms GraphM for different cases because of much smaller data access cost. The CPU utilization of GridGraph-M is 36%-45% (41% on average) for W_1, 39%-49% (45% on average) for W_2, 34%-42% (36% on average) W_3, 35%-47% (38% on average) for W_4, 24%-38% (28% on average) for W_5, 32%-41% (39% on average) for W_6, 29%-43% (35% on average) for W_7, and 27%-36% (31% on average) for W_8, respectively. We can observe that the characteristics of the workloads have impact on the performance of GraphSO and the higher performance can be obtained by GraphSO when the CPU utilization of a workload (e.g., W_5) is lower (because the ratio of data access time to the execution time is higher for this workload). Besides, we can see that GrahphSO performs better when the frequency of the *non-all-active algorithm* in the workloads (e.g., W_4 and W_5) is higher.

TABLE 4
Execution time (in seconds) of GraphSO integrating with the other systems (i.e., X-Stream [13], GraphChi [14], DynamicShards [16], LUMOS [17], Graphene [24], and Wonderland [25])

	TW	FS	U7	UN	CW
X-Stream-M	1,191	1,642	2,031	2,582	38,182
X-Stream-O	626	656	725	922	10,435
GraphChi-M	1,532	2,043	2,628	3,109	62,210
GraphChi-O	684	973	1,095	1,216	14,467
DynamicShards-M	1,195	1,651	2,302	2,642	41,527
DynamicShards-O	613	864	938	1,152	13,931
LUMOS-M	265	421	905	1,071	9,525
LUMOS-O	161	275	554	637	3,026
Graphene-M	79	95	126	207	992
Graphene-O	57	75	83	146	674
Wonderland-M	204	365	756	829	7,842
Wonderland-O	152	206	427	538	2,894
CGraph	183	326	624	792	3,595

4.6 Impact of Ratio of Memory Used to Buffer Graph

Next, we evaluate the impact of utilizing different ratios of physical memory to cache graph data. Fig. 20 shows the corresponding results of the four jobs executed on various graphs. All the results are normalized to the case of 0% of memory used to cache graph data. An interesting phenomenon observed from the highest performance is achieved when the ratio is neither too small nor too large. When the ratio is too small, i.e., a small part of graph structure data is cached in memory, the repeated I/O traffic of the chunks in the high-frequency range cannot be fully reduced and the memory resources are underutilized. On the contrary, when the ratio is too large, a large number of memory resources are applied to cache the graph structure data. Thus, there is intense contention for maintaining the job-specific data and run-time data (e.g., $TChunk$ table) in the memory among concurrent jobs, degrading the performance in turn. Therefore, appropriate ratio shall be selected when the data size cached in the memory and other data needed be concurrent jobs approach the total memory budget.

4.7 Integrated with other Systems

Finally, we integrate GraphSO into the other popular out-of-core graph processing systems and then evaluate their performance, thereby showing its generality and effectiveness. Table 4 describes the total execution time when running the four jobs on various schemes of different systems. The experimental results show GraphSO achieves considerable speedups in comparison with GraphM over all of the datasets. In detail, GraphSO obtains the speedup of 1.9-4.1 times, 2.1-4.3 times, 1.9-2.9 times, 1.5-3.1 times, 1.3-1.5 times, and 1.3-2.7 times compared to GraphM when they are integrated with X-Stream, GraphChi, DynamicShards, LUMOS, Graphene, and Wonderland, respectively. Note that different graph processing systems usually obtain diverse speedups after integrating with GraphSO, because their data accessing time occupies different proportions of their total execution

time. In general, the higher speedup can be obtained by GraphSO when the proportion is higher in the original system, because more proportion of the data accessing time can be reduced by eliminating the overhead of loading and storing unnecessary graph structure data. We can observe that GraphSO outperforms CGraph when integrating it with LUMOS, Graphene, and Wonderland, respectively, because of the smaller data access cost ensured by GraphSO.

5 Related Work

5.1 Disk-Based Graph Processing Systems

GraphChi [14] tries to reduce the random disk accesses, while X-Stream [13] aims to obtain sequential I/O performance for loading edges. GridGraph [15] achieves better locality and less I/O operations through a streaming-apply model and a two-level hierarchical partition strategy. NXgraph [36] achieves better cache locality, less disk I/O traffic, and fully CPU parallelism through flexible update strategies. HUS-Graph [37] obtains efficient I/O performance via a hybrid update strategy. CLIP [38] supports the beyond-neighborhood accesses and loaded data reentry to reduce total disk I/O. Wonderland [25] uses user-defined graph abstraction to achieve fast convergence speed for graph applications. LUMOS [17] enables a subgraph in an iteration to be exploited proactively to reduce disk I/O operations. However, these studies are mainly proposed to efficiently serve the execution of a single job, which suffer from much redundant storage and access cost of graph data as running concurrent jobs on same graph. Unlike them, GraphSO can be integrated with these systems to enable the concurrent jobs executed on them to fully share the storage and accesses of the same graph data.

5.2 Concurrent Graph processing Systems

Seraph [18], [21] achieves lower memory consumption for concurrent jobs by decoupling the data model of graph computation. CGraph [19], [22] proposes a novel correlations-aware execution model to reduce the data access cost. GraphM [20] is a graph storage system that eliminates the redundant storage and access overhead to the same graph. However, these solutions adopt the static partition strategy and entirely load the static partition to exploit the sequential disk I/O for concurrent jobs. It causes much unnecessary I/O traffic when most vertices are inactive in the static partition, eventually resulting in a lower throughput for concurrent jobs. Compared with them, GraphSO can significantly and transparently reduce the I/O traffic of concurrent jobs by skipping the unnecessary graph data streams and efficiently utilizing the memory resources.

In addition, some systems are recently proposed for concurrent graph queries. Wukong [39] achieves highly concurrent and low-latency graph queries by leveraging a RDMA-based approach. Congra [40], [41] proposes a dynamic graph queries scheduler to improve system throughput and resource efficiency. Nevertheless, these systems are not efficient for concurrently serving iterative graph processing jobs because of the different traversal characteristics among iterative graph processing and graph quires. In detail, the former usually requires to frequently process the entire graph, whereas the latter usually only handle different small parts of the graph.

5.3 Graph Storage Schemes

Some graph storage systems are drafted to accelerate out-of-core iterative graph processing by efficiently using fast storage devices. GraphOne [42] adopts a unified graph data store to efficiently support both graph databases and analytics engines. For less data access cost, TurboGraph [32] designs a pin-and-slide model to identify the corresponding graph data of the query vertices and pin these data in buffer pool. *Renen et al.* [43] optimizes the bandwidth utilization via a novel DRAM-resident buffer manager. FlashGraph [31], Graphene [24], and G-Store [28] are designed to efficiently support graph processing on SSD and also propose several schemes (such as the skipping of inactive vertices, graph data caching policy, and slide-cache-rewind scheduling) to reduce data access cost. However, these systems cannot efficiently handle concurrent graph processing jobs, because many redundant graph data are loaded and cached in the memory by them. Compared with them, GraphSO enables the data accesses to be shared by different jobs and also achieves the higher utilization rate of the loaded graph data.

5.4 Graph Partitioning Strategies

GraphChi [14] and VENUS [44] employ edge-cut partition scheme for parallel processing and better locality. X-Stream [13] and NXgraph [36] adopt vertex-cut partition scheme to partition a graph for balanced load between partitions. Meanwhile, some dynamic graph partitioning strategies are also designed to reduce I/O traffic. *Luis et al.* [45] proposes an adaptive partitioning for dynamic graph processing for better performance. DynamicShards [16] employs dynamic partitions to eliminate unnecessary I/O for loading the useless edges from partitions. However, these strategies are drafted to serve the execution of a single job. When using them to support the execution of concurrent jobs on the same graph, they face much unnecessary repartitioning cost. The repartitioning scheme of GraphSO enables different jobs to share the repartitioned results, thus amortizing the repartitioning overhead among these jobs.

6 Conclusion

This paper proposes GraphSO, an efficient structure-aware graph storage system, which can be integrated with existing out-of-core graph processing systems to efficiently execute concurrent iterative graph processing jobs. Specifically, GraphSO can judiciously load the required graph data to efficiently construct the logical partitions for the execution of the concurrent jobs, thereby reducing much unnecessary I/O overhead and maximizing the utilization of the loaded graph data. Moreover, GraphSO designs a buffering strategy to efficiently use the memory resources so as to minimize the I/O traffic. Experimental results show that GraphSO can be effectively integrated with existing out-of-core graph processing systems and achieve a significant throughput improvement for concurrent jobs, in comparison with state-of-the-art approaches. In the future, we will research how to optimize our approach to efficiently utilize the SSDs to serve the execution of many concurrent graph processing jobs.

References

[1] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[2] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010, pp. 135–146.

[3] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. J. Lin, "Graphjet: Real-time content recommendations at twitter," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1281–1292, 2016.

[4] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang, "Knightking: a fast distributed graph random walk engine," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 524–537.

[5] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 301–316.

[6] "facebook," http://www.facebook.com/, 2020.

[7] "Google," http://www.google.com/, 2020.

[8] "twitter," https://www.twitter.com/, 2020.

[9] "tencent," https://www.tencent.com/en-us/about.html/, 2020.

[10] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford Digital Library Technologies Project, Tech. Rep., 1998.

[11] S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (scc) in small-world graphs," in *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–11.

[12] S. Gupta, R. Kumar, K. Lu, B. Moseley, and S. Vassilvitskii, "Local search methods for k-means with outliers," *Proceedings of the VLDB Endowment*, vol. 10, no. 7, pp. 757–768, 2017.

[13] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 472–488.

[14] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, 2012, pp. 31–46.

[15] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proceedings of the 2015 USENIX Annual Technical Conference*, 2015, pp. 375–386.

[16] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic i/o optimization for disk-based graph processing," in *Proceedings of the 2016 USENIX Annual Technical Conference*, 2016, pp. 507–522.

[17] K. Vora, "LUMOS: dependency-driven disk-based graph processing," in *Proceedings of the 2019 USENIX Annual Technical Conference*, 2019, pp. 429–442.

[18] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: an efficient, low-cost system for concurrent graph processing," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, 2014, pp. 227–238.

[19] Y. Zhang, X. Liao, H. Jin, L. Gu, L. He, B. He, and H. Liu, "CGraph: A correlations-aware approach for efficient concurrent iterative graph processing," in *Proceedings of the 2018 USENIX Annual Technical Conference*, 2018, pp. 441–452.

[20] J. Zhao, Y. Zhang, X. Liao, L. He, B. He, H. Jin, H. Liu, and Y. Chen, "GraphM: an efficient storage system for high throughput of concurrent graph processing," in *Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 3:1–3:14.

[21] J. Xue, Z. Yang, S. Hou, and Y. Dai, "Processing concurrent graph analytics with decoupled computation model," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 876–890, 2017.

[22] Y. Zhang, J. Zhao, X. Liao, H. Jin, L. Gu, H. Liu, B. He, and L. He, "CGraph: A distributed storage and processing system for concurrent iterative graph analysis jobs," *ACM Transactions on Storage*, vol. 15, no. 2, pp. 10:1–10:26, 2019.

[23] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, 2012, pp. 17–30.

[24] H. Liu and H. H. Huang, "Graphene: Fine-grained IO management for graph computing," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, 2017, pp. 285–300.

[25] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," in *Proceedings of the 23th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 608–621.

[26] U. Meyer, "Single-source shortest-paths on arbitrary directed graphs in linear average-case time," in *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2001, pp. 797–806.

[27] X. Li, M. Zhang, K. Chen, and Y. Wu, "Regraph: A graph processing framework that alternately shrinks and repartitions the graph," in *Proceedings of the 32nd International Conference on Supercomputing*, 2018, pp. 172–183.

[28] P. Kumar and H. H. Huang, "G-store: high-performance graph store for trillion-edge processing," in *Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 830–841.

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

[30] Y. Zhang, X. Liao, H. Jin, L. Gu, G. Tan, and B. B. Zhou, "Hotgraph: Efficient asynchronous processing for real-world graphs," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 799–809, 2017.

[31] D. Zheng, D. Mhembere, R. C. Burns, J. T. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, pp. 45–58.

[32] W. Han, S. Lee, K. Park, J. Lee, M. Kim, J. Kim, and H. Yu, "Turbograph: a fast parallel graph engine handling billion-scale graphs in a single PC," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013, pp. 77–85.

[33] "Law." http://law.di.unimi.it/datasets.php, 2021.

[34] "Snap." http://snap.stanford.edu/data/index.html, 2021.

[35] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[36] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "Nxgraph: An efficient graph processing system on a single machine," in *Proceedings of the 2016 IEEE International Conference on Data Engineering*, 2016, pp. 409–420.

[37] X. Xu, F. Wang, H. Jiang, Y. Cheng, D. Feng, and Y. Zhang, "Husgraph: I/o-efficient out-of-core graph processing with hybrid update strategy," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 3:1–3:10.

[38] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, "Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o," in *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017, pp. 125–137.

[39] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li, "Fast and concurrent RDF queries with rdma-based distributed graph exploration," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 317–332.

[40] P. Pan and C. Li, "Congra: Towards efficient processing of concurrent graph queries on shared-memory machines," in *Proceedings of the 2017 International Conference on Computer Design*, 2017, pp. 217–224.

[41] P. Pan, C. Li, and M. Guo, "Congraplus: Towards efficient processing of concurrent graph queries on NUMA machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 1990–2002, 2019.

[42] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," in *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, 2019, pp. 249–263.

[43] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato, "Managing non-volatile memory in database systems," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1541–1555.

[44] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. S. Lui, and C. He, "VENUS: vertex-centric streamlined graph computation on a single PC," in *Proceedings of the 31st IEEE International Conference on Data Engineering*, 2015, pp. 1131–1142.

[45] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "Adaptive partitioning for large-scale dynamic graphs," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, 2013, p. 35.