Improving Graph Compression for Efficient Resource-Constrained Graph Analytics

Qian Xu
Renmin University of China
xuqian@ruc.edu.cn

Juan Yang
Beijing HaiZhi XingTu Technology Co., Ltd
yangjuan@stargraph.cn

Feng Zhang*
Renmin University of China
fengzhang@ruc.edu.cn

Zheng Chen
Renmin University of China
chenzheng123@ruc.edu.cn

Jiawei Guan
Renmin University of China
guanjw@ruc.edu.cn

Kang Chen
Tsinghua University
chenkang@tsinghua.edu.cn

Ju Fan
Renmin University of China
fanj@ruc.edu.cn

Youren Shen
Beijing HaiZhi XingTu Technology Co., Ltd
shenyouren@stargraph.cn

Ke Yang
Beijing HaiZhi XingTu Technology Co., Ltd
yangke@stargraph.cn

Yu Zhang
Renmin University of China
yu-zhang21@ruc.edu.cn

Xiaoyong Du
Renmin University of China
duyong@ruc.edu.cn

ABSTRACT

Recent studies have shown the promise of directly processing compressed graphs. However, its benefits have been limited by high peak-memory usage and unbearably long compression time. In this paper, we introduce Laconic, a novel rule-based graph processing solution that overcomes the challenges of restricted memory and impractical compression time faced by existing approaches. Laconic, for the first time, ensures minimal memory overhead during compression and significantly reduces graph sizes, thus reducing peak memory demand during computations. By employing an efficient parallel compression algorithm, Laconic achieves a remarkable reduction in compression time. In our experiments, we compare Laconic with state-of-the-art solutions. The results demonstrate that Laconic outperforms other methods, reducing peak memory consumption by an average of 70% during compression and 66% during computation. Additionally, Laconic reduces rule compression time by an average of 93% compared to traditional rule-based compression, achieving a 2.47× higher compression ratio, and providing a 2.12× performance speedup.

PVLDB Reference Format:

Qian Xu, Juan Yang, Feng Zhang, Zheng Chen, Jiawei Guan, Kang Chen, Ju Fan, Youren Shen, Ke Yang, Yu Zhang, and Xiaoyong Du. Improving Graph Compression for Efficient Resource-Constrained Graph Analytics. PVLDB, 17(9): 2212 - 2226, 2024.

doi:10.14778/3665844.3665852

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/xuqianmamba/Laconic.

*Feng Zhang is the corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 17, No. 9 ISSN 2150-8097.
doi:10.14778/3665844.3665852

1 INTRODUCTION

The ever-increasing graph sizes have fueled a surge in graph compression research [5, 9, 10, 12, 22, 25, 31, 49, 69, 70, 90, 91, 94, 96]. Recent studies have highlighted the potential of direct processing on compressed graphs [13, 25], yielding promising results in terms of enhancing graph analytics efficiency. However, despite the efficiency advantages inherent in these methods, a notable oversight persists concerning the consideration of memory consumption and compression time. Typically, these methods demand several times the original graph's size in memory resources and necessitate extended compression time. For example, Figure 1 illustrates the memory consumption of the state-of-the-art in-memory graph computing system Ligra [91] and its implementation with a compression module, Ligra+ [92], during the execution of the PageRank computation and compression tasks on the 28.65 GB *uk-2007-05* graph, respectively. Although Ligra+ effectively reduces the size of the graph, there is no significant decrease in peak memory usage, which is 121.85GB, 4.2 times the size of the original graph, during the transition from compression to computation. Moreover, Ligra+ needs 605 seconds to compress *uk-2007-05* graph, which is unbearable in practice. Similar limitations also exist in CompressGraph [20], the state-of-the-art rule-based graph compression framework, whose peak memory usage is 65.38 GB and compression time is 700 seconds handling *uk-2007-05*. This resource overhead hinders their broad adoption. In order to fully harness the potential of direct computation techniques, there is an urgent need for an effective solution that concurrently minimizes memory utilization and compression time.

Direct processing compressed graphs in resource-constrained environment yields three compelling advantages. First, performing computations directly on compressed data has been demonstrated to significantly enhance graph analytics performance [3, 34, 53, 74, 75]. On one hand, analytical algorithms can leverage the precomputed results from redundant data, thereby reducing computational overhead. On the other hand, compressed graphs can be stored

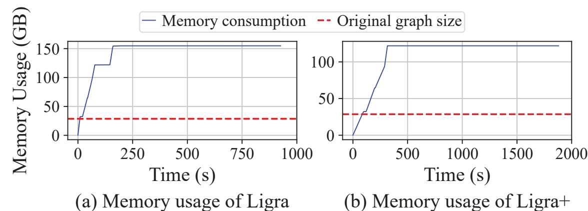


Figure 1: Memory consumption on *uk-2007-05* during graph processing.

more efficiently in CPU caches, thus improving data locality. Second, compressed graph data structures typically demand substantially less memory compared to their uncompressed counterparts. When appropriately designed, these structures enable the analysis of larger graphs on systems that would otherwise be constrained by memory limitations, thereby improving scalability. For example, Ligra+ needs more than 600GB memory to manage *uk-2014* dataset, while we can reduce the peak memory consumption to less than 300GB memory with no performance loss. Third, storing compressed graph data occupies less disk space, which can result in reduced storage costs, especially for applications dealing with large-scale graph datasets. For example, upon applying our compression technique, the *uk-2007-05* dataset, originally occupying 28.65 GB, is reduced to a mere 3.48 GB of storage space (detailed in Section 5.4).

Enabling direct processing of compressed graphs in resource-constrained environments presents three key challenges. First, both the compression and computation processes necessitate the allocation of memory for auxiliary structures, introducing complexity into memory optimization. Second, the intricate balance between minimizing memory usage and maximizing computational efficiency presents a significant challenge. High compression ratios achieved by certain graph compression methods often lead to notable compression time overhead, whereas the use of lightweight compression methods may compromise compression effectiveness, resulting in increased peak memory consumption. Third, while rule-based compression and reference compression [10] (a technique that identifies and encodes the common neighbors of two vertices using reference coding) algorithms achieve a reasonable trade-off between compression ratio and graph processing efficiency, they often result in prolonged compression durations and increased memory usage. The reduction of compression overhead through meticulous design remains a challenging task.

Many compression methods have been proposed in recent years [1, 4–6, 10, 12, 15, 20, 25, 27, 28, 32, 46, 51, 59, 72, 73, 79, 80, 83, 86, 88, 92, 98–100, 112–114, 116]. However, none has successfully addressed the aforementioned challenges. For example, Brisaboa et al. [12] proposed a method based on k^2tree to store the adjacency matrix, which effectively reduces redundancy in the graph. However, the compression based on the adjacency matrix does not yield a favorable compression ratio. Moreover, Ligra+ [92] is a novel framework that explores high-speed processing on compressed graphs using encoding-based compression in the adjacency list. Unfortunately, it does not effectively reduce the size of the vertices and edges in the graph, resulting in high peak memory usage during compression and computation. Recently, numerous rule-based compression works, such as TADOC [116], have demonstrated promising prospects for reducing redundancy and enabling computation without decompression. However, rule-based compression incurs significant time and space overhead during compression, making it unsuitable for resource-constrained application scenarios.

We present a low-overhead, high-ratio, and expressive compression engine, Laconic, which effectively addresses these challenges. First, we propose an adaptive block hybrid compression based on encoding and rule compression, facilitating data partitioning at arbitrary granularities. Laconic can handle a wide range of graphs through fine-grained data segmentation, limiting the peak memory consumption while ensuring algorithm flexibility. Second, directly fusing the encoding- and rule-based compression can incur low compression ratios. Therefore, we incorporate two sub-modules, including a locality recovery submodule and a parallel encoding compression submodule, between the steps of the hybrid compression method to further optimize compression. This approach effectively reduces memory and time overhead in graph computations while eliminating redundancy. Third, we introduce a parallel compression algorithm specifically designed for time-consuming rule-based compression modules, which significantly reduces compression time while adhering to peak memory limitations.

We use five graph algorithms with 12 diverse graphs for evaluation. Experiments show that compared to the state-of-the-art solutions, Laconic reduces averagely 70% and 66% memory consumption during compression and computation, respectively. Meanwhile, Laconic reduces 93% compression time on average while achieving over 2.47× compression ratio with 2.12× performance speedup.

We make three primary contributions in this paper.

- We propose Laconic, a compressed graph computation engine capable of performing graph compression and computation in memory-constrained environments, requiring at most 80% the size of the graph.
- We introduce a compression approach that combines parallel rule compression with encoding compression, achieving an 11× improvement in compression speed and a 58% increase in compression ratio compared to the state-of-the-art rule-based compression methods.
- We conduct comprehensive experiments to demonstrate the performance benefits of Laconic over the state-of-the-art solutions.

2 BACKGROUND

Graph compression [5, 9, 10, 12, 20, 22, 25, 31, 49, 69, 70, 90, 91, 94] plays a crucial role in reducing the overhead of graph analytics by reducing the graph size.

2.1 Existing Graph Compression

In the context of graphs, redundancies refer to repeated information, such as common neighbors within the graph structure. Redundancies exist in a wide range of graph applications [7, 36, 41, 54, 84, 106]. These redundancies can lead to inefficiencies in storage, processing, and analytics of the graph data.

Utilizing compression to process graph data can efficiently eliminate these redundancies. Most graph compression schemes a based on two graph representations – adjacency matrix [12, 31, 93, 94] and adjacency list [9, 10, 20, 22, 92]. Although they have go compression ratios, the compressed result of the adjacency list more convenient for subsequent graph analytics than the adjacen matrix format. The reason is that, in graph computing, visiting neighbors of a single vertex is the most basic operation, which more suitable for an adjacency list. We next show current popul graph compressions, including encoding-based compression a vertex reuse compression.

Encoding-based compression. Encoding-based compressi methods encode vertices with few codes, thus achieving significa memory savings. Among them, we utilize *variable length codi* and *delta encoding* in Laconic.

Variable length coding methods change the fixed-length rep sentation of vertices in the adjacency list. Some methods, such as Huffman coding [28, 51, 92], adjust the cost of storing different vertices based on their occurrence frequency to reduce storage overhead. Other methods target different data characteristics. For example, Elias Gamma encoding and Elias Delta encoding [32] are suitable for compressing small numbers. Null Suppression [4] achieves compression by removing leading zeros in elements.

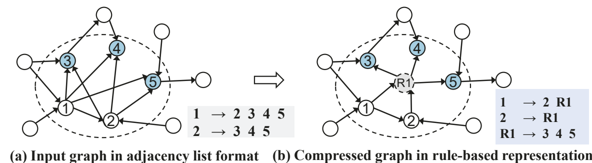
Delta encoding [86] changes the original graph data by encoding the number as the difference between its value and the previous value. By combining variable-length coding, data size can be significantly reduced to achieve high compression ratios.

For the other methods, Run length encoding [1] represents continuously repeated intervals in the adjacency list as a regular coding form. Bitmap encoding [27, 98, 99] is suitable for graphs whose data values have only a few possibilities. It can be mapped to intervals with low encoding costs.

Rule-based compression. Rule-based compression is a data compression technique that entails identifying recurring patterns within the data and encoding them as rules. Here, a "rule" is characterized as a consecutive sequence of neighbors with a length of two or more that is repeated multiple times.

Rule-based compression has been demonstrated as a major accomplishment in the field of data science [25, 79, 80, 112–114, 116]. CompressGraph [20] represents the state-of-the-art approach in the field of graph rule compression. It leverages rule vertices to replace repeated adjacent edges in adjacency lists. This method not only effectively eliminates redundancy within the graph, but also demonstrates promising results in processing compressed data directly, thus bypassing the need for decompression.

Example. Figure 2 illustrates a rule-based compression instance of CompressGraph, where each number represents a vertex in the graph. Figure 2 (a) shows the adjacency list of the graph, where vertices 1 and 2 have a common set of repeated neighbors (vertices 3, 4, and 5). CompressGraph is capable of capturing such redundancy by encoding these repeated neighbors as virtual rule vertices, reducing the representation of duplicated edges. As shown in Figure 2 (b), the set of recurring neighbors {3, 4, 5} is represented as rule R1. Since rule compression does not disrupt the connectivity of the graph, CompressGraph modifies graph algorithms to ensure that most algorithms, such as BFS and PR, can operate correctly within the compressed graph, thereby avoiding decompression overhead.



(a) Input graph in adjacency list format (b) Compressed graph in rule-based representation

Figure 2: An example of rule and rule-based compression.

2.2 Trade-off Compression Ratio and Time

Compression and decompression time plays an important role among graph systems. Current graph compression techniques [5, 9, 10, 12, 22, 25, 31, 49, 69, 70, 90, 94] offer high compression ratios, but the encoding and decoding processes of heavyweight compression algorithms are excessively complex, resulting in significant time overhead. Graph analytics systems [18, 21, 33, 35, 47, 52, 56, 67, 71, 77, 85, 91, 103, 105, 111, 118–120] often require real-time processing and low latency, and cannot tolerate prolonged delays. Upon exploring *heavyweight compression algorithms* in graph analytics systems, we observe that while they can achieve higher compression ratios, they are accompanied by longer compression and decompression time, which do not significantly enhance the overall performance and stability of the system.

Lightweight compression [1, 4, 27, 32, 83, 86, 89, 98, 99] is a trade-off approach between compression ratio and compression time. The compression methods used in lightweight compression are relatively simple. In comparison to heavyweight compression algorithms, they sacrifice higher compression ratios in favor of faster compression and decompression.

In recent years, rule-based compression algorithms have been proven capable of performing direct computations and queries [20, 25, 112–114, 116] without decompression, making rule compression unnecessary to decompress after reduction. However, the initial compression time of rule-based compression is still too long, even hundreds of times the graph computation time, not to mention the high memory overhead.

3 REVISITING PREVIOUS GRAPH COMPRESSION SOLUTIONS

In this part, we revisit previous graph compression techniques to explain the rationale behind our novel graph compression design. Our approach focuses on achieving low compression peak memory, low computational peak memory, and minimized compression time.

Why compress graphs before computation? For small graphs in graph computing tasks, the most efficient and cost-effective approach is to directly load the graph into memory and perform computations. However, with the growth of the graph size and auxiliary memory overhead during computation, processing large graphs directly in memory necessitates a substantial memory allocation. The mainstream methods to address this problem include 1) in-memory compression [5, 9, 10, 12, 22, 25, 31, 49, 69, 70, 90, 91, 94], 2) disk-based computation [21, 35, 56, 71, 77, 120], and 3) distributed

computation [52, 76, 118, 119]. Disk-based computation requires minimal hardware and maintenance overhead, but it suffers from high I/O time overhead. Conversely, distributed computation offers excellent time performance but demands higher development and maintenance costs. In contrast, compression before computation can significantly reduce the size of the graph, decreasing or avoiding I/O overhead during subsequent computations, while keeping costs low.

Why don't existing compression algorithms for graphs apply? Unlike other compression tasks, graph compression for computation demands three essential principles: low peak memory usage, low compression time, and low computation overhead. Existing compression algorithms face several limitations. First, algorithms with high compression ratios such as WebGraph [5, 10] and K^2-tree [12] are heavyweight, increasing the computational burden during graph processing due to their high decompression overheads. Second, lightweight compression algorithms such as RLE [1], delta [86] exhibit low compression efficacy. Third, while rule-based compression has been proven to accomplish graph computation tasks without decompression, it requires high peak memory usage as it needs to store a vast number of intermediate results, and it has a high compression complexity. Ensuring low spatiotemporal overheads for both compression and computation has become a primary reason limiting the deployment of compression algorithms in real-world graph applications.

4 SYSTEM DESIGN

4.1 Overview

We present an overview of Laconic in Figure 3. Unlike other graph compression systems, Laconic takes both the original graph $G(V, E)$ in the CSR format [107] and the available memory M for compression as input. The output is the compressed CSR result $G(V_{compress}, E_{compress})$, similar to previous graph compression systems [10, 20, 91].

Modules. Laconic consists of three main modules: 1) memory-aware adaptive graph slicing module, 2) parallel rule-based subgraph compression module, and 3) graph expression recovery module. The memory-aware adaptive graph slicing module adaptively partitions the original graph based on the relationship between the input memory and the graph. The parallel rule-based subgraph compression module effectively compresses each sliced subgraph, eliminating redundancies in the original graph. The graph expression recovery module includes three main steps: compressed subgraph merging, vertex reordering, and parallel encoding compression. These steps collectively augment the expressive capacity of the compressed graph while reducing peak memory usage during computation.

Novelties. Laconic introduces several novel aspects. First, Laconic employs a memory-aware adaptive graph-slicing approach before compression. This reduces memory usage during the compression process and increases the maximum graph size that the system can handle. Different from traditional graph analytics methods, Laconic considers the additional space overhead during rule compression and provides a customizable configuration, allowing for precise control over the peak memory usage of the compression process. Second, Laconic presents a parallel rule-based subgraph compression method that accelerates the otherwise time-consuming rule-based compression process. This represents the first parallelized rule-based graph compression work. Third, Laconic uses a graph expression recovery module with a hybrid compression scheme that combines rule-based compression with other compression methods. This approach effectively utilizes redundancies at different levels within the graph and resolves conflicts between the two compression methods. This results in higher compression ratios and lower memory and time overhead, which, to the best of our knowledge, is the first work to integrate rule-based compression with other compression methods efficiently in memory constrained environment.

4.2 Memory-Aware Adaptive Graph Slicing Module

In this part, we present Laconic's memory-aware adaptive graph-slicing module in Figure 3. First, Laconic calculates the number of subgraphs and their sizes required to complete the graph analytics task under the constrained memory size provided by the user. Second, to maintain graph locality, ensure the effectiveness of subsequent compression, and avoid introducing edges that connect different subgraphs during the slicing process, we choose to partition vertices and their neighbors into different subgraphs based on the IDs in the original graph's CSR_{vlist}. We show an example in Figure 3 (a). The edge marked by the dashed line shall be partitioned into the same subgraph as its source vertex. This is because this edge is only recorded in the adjacency list of the source vertex, offering the benefit of preventing edges from crossing different subgraphs.

To ensure that the partitioned subgraphs do not exceed the given memory limit during compression, the key is to find the appropriate subgraph size. Subgraphs that are too small cannot capture the redundancy in the graph, leading to poor compression ratios. Conversely, subgraphs that are too large result in excessively high peak memory during compression. Next, we demonstrate the maximum subgraph size for which the compression task can be performed stably.

Peak memory guarantee. In a graph without duplicate edges, when the neighbors of a vertex form a specific permutation $a_1, a_2, \ldots, a_n$, the theoretical upper limit on the number of rules during the compression process is the number of all its subsets with a size greater than 1. Moreover, storing each subset of size $|S|$ requires an auxiliary space of size $|S|$. Therefore, the additional memory required to store all possible rules is $\sum_{|R|=2}^{|V|}(R \cdot C_{|V|}^{|R|})$, where $|V|$ represents the number of vertices in the graph, and $|R|$ denotes the size of the rules. When the number of vertices $|V|$ is equal to 20, the peak memory can reach 40 MB. When the number of vertices $|V|$ is equal to 30, the peak memory can reach 60 GB. Although real-world graphs typically do not contain all possible rules, existing rule-based compression methods often encounter scenarios where the memory required to store the rules far exceeds that required to store the original graph itself. Therefore, limiting the size of the rules to reduce the peak memory during compression becomes both urgent

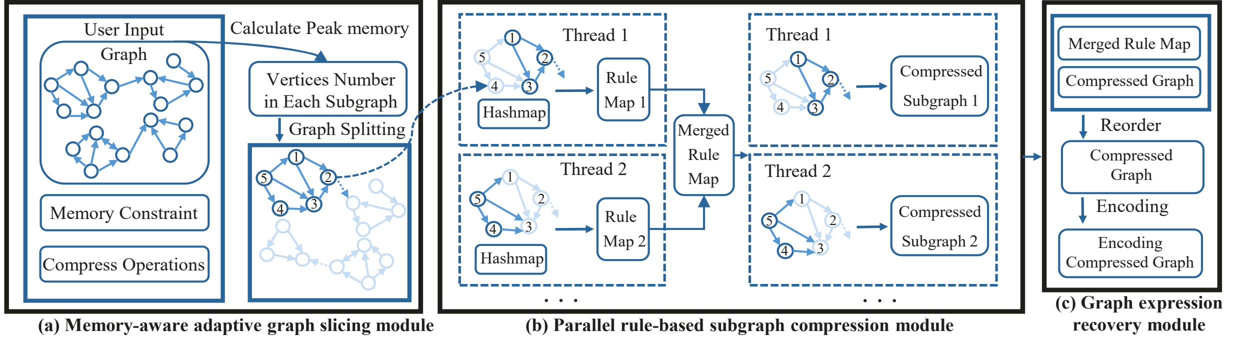


Figure 3: Laconic overview.

and important. We find that the peak memory can be significantly reduced by satisfying the following three requirements:

- Retaining only the smallest rules of size 2 (we only count the occurrence frequency of two adjacent elements within a vertex's neighbors) in each compression.
- Ensuring the orderliness of vertex neighbors.
- Considering only two consecutive elements that appear in a vertex's adjacency list as legal rules.

We introduce a concept called "gap" as the difference between two adjacent neighboring elements in a pair, and the term "gaps" refers to the total sum of differences between two adjacent neighboring elements in all pairs in our proof. For instance, if a vertex's adjacency list is {1,4,5,6}, then the "gaps" within this list would be 3+1+1=5, while the "gap" between the neighboring element pair {1,4} being 3. It can be observed that in an ascending adjacency list, the "gaps" shall not exceed |V|-1, where |V| represents the total number of vertices in the graph.

Proof. We demonstrate that by limiting the size of the partitioned sub-graphs, we can constrain the peak memory consumption during the rule compression process. Assuming the graph has $|V|$ vertices, among all possible rules, there are at most $|V|-1$ rules with a "gap" of 1: {0,1},{1,2}...{$|V|-2$, $|V|-1$}. There are at most $|V|$-2 rules with a gap of 2, and so on. Therefore, the number of all rules will not exceed $\frac{|V|^2}{2}$. Furthermore, we partition the original graph into multiple subgraphs with the same number of vertices to further reduce the memory required to store the rules. For example, assuming we set the number of vertices in each subgraph as K, in this case, the "gaps" that these K adjacency lists can provide is at most $O(K|V|)$. Moreover, even if the rules appear in the order of increasing "gap", the total number of rules can be at most $\sum_{gap=1}^{\sqrt{2K}} gap \cdot (|V|-gap)$ (this is because $1 + 2 + \ldots + \sqrt{2K} = K$). Thus, for a specific value of K, the maximum number of rules generated is only $O(\sqrt{2K} * |V|)$. Following the aforementioned analysis, we can derive the following lemma:

Lemma 1. *By adjusting the size of the divided subgraph $G_{sub}(|V_{sub}|, |E_{sub}|)$ to satisfy $O(\sqrt{|V_{sub}|} \cdot |V| + |E_{sub}|) \leq M$, where M represents the available memory, we can complete the graph compression under memory constrained environment.*

Insight. It has been observed that the upper bound on the sum of differences in all pairs of neighboring elements (referred to as "gaps") in a sorted adjacency list of a vertex does not exceed N-1. Laconic's memory-aware adaptive graph slicing scheme has two major innovations. First, we use this upper bound to deduce the maximum number of binary rules that can occur within a given "gap", thereby reducing the peak memory for rule compression from the exponential level of the vertex to a polynomial level, specifically bringing it down to a ($O(\sqrt{2K} * |V|)$) level. Second, since the selection of the parameter K is configurable, we can ensure complete control over the peak memory of the compression process by adjusting the parameter K.

4.3 Parallel Rule-Based Subgraph Compression Module

4.3.1 ***General Design.*** Laconic's rule-based parallel subgraph compression module differs from traditional rule-based compression in two ways. First, for the consideration of low peak memory usage, Laconic only captures rules of size 2 in each iteration. To capture larger rules, Laconic needs multiple iterations of parallel rule compression. Second, during rule capturing and frequency counting, Laconic does not maintain a global rule list for the incoming graph, as doing so would incur significant synchronization overhead.

We show an instance in Figure 3 (b). Laconic's parallel rule-based subgraph compression module processes the subgraphs received after graph partitioning. While the previous graph partitioning in Section 4.2 is primarily aimed at preventing the graph's peak memory from exceeding the machine's memory capacity, Laconic goes a step further to accelerate the rule-based compression. In this process, Laconic further subdivides the input subgraphs. For the given example in Figure 3 (b), the input subgraph is split into two parts: vertices 1, 2, 3 and their outgoing edges are assigned to thread 1 for compression, whereas vertices 4, 5 and their outgoing edges are assigned to thread 2 for compression. After compression, Laconic merges the compressed subgraphs.

4.3.2 ***Example.*** In this section,we introduce the differences between Laconic's rule-based compression module and traditional rule-based compression, and illustrate the contrast between them

using an example. Laconic's rule-based compression module differs from traditional rule-based compression in two main aspects. First, traditional rule-based compression adopts a global traversal approach to mine rules. However, this increases the complexity of rule capture and replacement. For instance, when we plan to replace {1, 2} in {0, 1, 2, 3, 4} with $\{R_1\}$, traditional rule-based compression not only needs to modify the occurrence counts of {0, 1} and {2, 3}, but also decreases the occurrence counts of all rules that include {1} and {2}, such as {1, 2, 3, 4}. In contrast, Laconic employs a progressive rule traversal approach, which captures only binary rules. A binary rule is two consecutive vertices that frequently appear in the adjacency list. Second, in typical scenarios, traditional rule traversal replaces the most frequent rule sequentially. This restriction affects the parallelism of the rule compression process. On the other hand, Laconic's progressive rule traversal can traverse all frequent rules in a single round. We provide an example in Figure 4 to illustrate the differences between the progressive rule traversal and the traditional rule traversal. Figure 4 (a) and Figure 4 (b) show the original graph and its adjacency list format, while Figure 4 (c) demonstrates the rule compression process based on progressive rule traversal and traditional traversal.

As shown in Figure 4 (c), using traditional rule traversal, the first batch of replacement rules is just {0, 1}, and the second batch is just {4, 5}. In contrast, progressive rule traversal allows all binary rules to be added to the rule capture queue at once, regardless of their frequency. This approach captures multiple rules at a time in any compression snapshot. In this example, at the first level, the captured rules are {0, 1}, {1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}, and {6, 7}. In the second level, two new rules $\{R_1, R_3\}$ and $\{R_4, R_6\}$ are captured. This significantly reduces the number of rule capture rounds, and since multiple rules can be replaced in each round, we can assign the rules to be replaced to different threads, achieving parallelization while avoiding conflicts caused by different threads replacing the same rule.

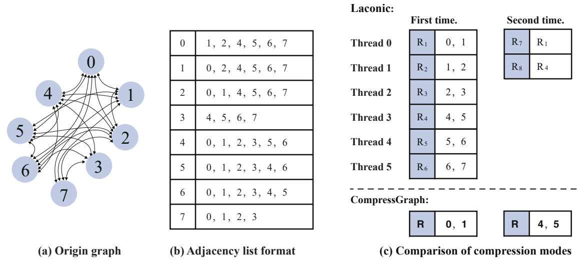


Figure 4: Rule generation comparison between Laconic and traditional rule-based compression.

4.3.3 *Algorithm Details.*

We illustrate the algorithm of parallel rule-based subgraph compression in Algorithm 1. Specifically, upon receiving the input graph, we first partition it into subgraphs for overall initialization. At the beginning of each iteration, we clear *Buckets* generated from the previous iteration. We design three stages of parallelism and synchronization mechanisms for each iteration. The first parallel stage (lines 13-18) counts all adjacent pairs of in the input adjacency list and their occurrences. The second parallel stage (lines 19-23) performs preliminary filtering on these binary elements to exclude infrequent rules (i.e., rules that occur less than a certain number of times, default being 4) and forms preliminary rules. The third parallel stage (line 9) conducts rule replacement on the input graph based on the generated rules in parallel. After completing the three parallel stages, we pass the iteration results to the next iteration to complete the entire iteration process. After completing all iterations, we first apply the filtering of infrequent rules to all subgraphs and then merge the individual subgraphs to achieve compression for the entire graph.

Hash_Pair_to_Buckets is the first parallel stage of the algorithm. Upon receiving the incoming uncompressed graph, we partition it into equally sized subgraphs and assign each subgraph to different threads. We use hashing to map each adjacent pair of elements in each subgraph to different buckets. We ensure that any adjacent pair of elements does not appear in two or more buckets, avoiding write conflicts during parallel frequency counting of element pairs.

Filter_and_Assign_Rule_IDs is the second parallel stage of the algorithm. We assign the generated buckets from the first parallel stage to different threads. This process consists of two steps. First, we count the occurrences of all element pairs in the buckets and remove all pairs with frequencies below the predefined *Frequent_Threshold*. Second, we traverse all element pairs in the buckets and assign rule IDs to each of them.

Find_and_Replace_Rule is the third parallel stage of the algorithm. We reassign the subgraphs partitioned during the first parallel stage and the hashed buckets processed during the second stage to different threads. Each subgraph is traversed using a sliding window approach, and rule replacements are performed using regular expression matching.

Algorithm 1: Parallel rule-based subgraph compression

```
1  Function lock_free_rule_based_compression(G = (V, E),
     iterations_num γ, frequent_threshold δ, thread_num λ):
2      subgraphs[] ← split_graph(G)
3      buckets ← create a mapping from a pair to 0..λ
4      for i ← 0 to γ do
5          init_hash_buckets(buckets)
6          for sg in subgraphs do
7              hash_pair_to_buckets(sg, buckets)
8              filter_and_assign_rule_ids(buckets, δ)
9              sg ← find_and_replace_rules(sg, buckets)
10     filter_infreq_rules(subgraphs)
11     CG ← merge_all_subgraphs(subgraphs)
12     return CG
13 Function hash_pair_to_buckets(subgraph, buckets):
14     for v in subgraph do
15         D ← degree[v]
16         adj ← adjacency list of v
17         for i ← 0 to D − 1 do
18             buckets.insert(pair_hash(adj[i], adj[i + 1]))
19 Function filter_and_assign_rule_ids(buckets, δ):
20     for bucket in Buckets do
21         count_freq_and_remove_infreq(bucket, δ)
22         for pair in bucket do
23             assign_id_to_rules(pair)
```

Filter_Infreq_Rules filters infrequent rules before merging all subgraphs. We traverse in parallel each subgraph obtained after the final iteration of compression and count the occurrences of rule vertices in the neighbors of the original graph. We then delete infrequent rules and restore their occurrences in the adjacency lists of all original vertices as neighbors of those rules.

Complexity analysis. Next, we analyze the complexity of each parallel stage in every iteration. (1) In the first parallel stage, a hash operation has to be performed on adjacent element pairs in each sub-graph, resulting in a complexity of *O(E-V)*. This is due to, in CSR representation, an adjacency list containing K elements having $K-1$ adjacent element pairs, and each vertex possessing one such adjacency list. (2) In the second parallel stage, all elements within the bucket need to be scanned twice: one to count the occurrences of all element pairs and the other to filter out infrequent rules and allocate rule IDs for frequent rules. Since the number of elements in the bucket is *O(E-V)*, this step's complexity is also *O(E-V)*. (3) In the third rule-replacement stage, all neighbors of every vertex need to be traversed. Pairs of neighbors appearing more than a given threshold are then replaced with their corresponding rule IDs. Both traversal and replacement operations have a complexity of *O(1)*, resulting in an overall complexity of *O(V+E)* for this process. (4) Finally, the rule-filtering procedure preceding the sub-graphs merge only requires a single traversal of all vertexes and edges in the sub-graphs. Similar to (3), but the replacement action changes from replacing vertex pairs with rules to restoring infrequent rules to vertex pairs. Thus, this step also has a time complexity of *O(V+E)*. The complexity for merging sub-graph stands at *O(1)*. Consequently, the overall time complexity of the algorithm is *O(T(V+E))*, where T denotes the number of algorithm iterations.

4.3.4 ***Theoretical Guarantee***.

In this part, we provide a theoretical guarantee of the compression ratio for the parallel rule-based compression algorithm and demonstrate its correctness in parallelization.

Validity. Estimating the compression ratio for rule-based compression poses a challenging task. All rule-based compression algorithms aim to solve the problem of obtaining the minimal context-free grammar that generates a given string. However, finding the optimal solution is not straightforward due to the NP-completeness of the decision version of this problem [16].

In parallel rule-based compression, when multiple rules are replaced simultaneously, the previously replaced rules can affect the occurrence frequency of other rules, as shown in Figure 4 (c). During parallel graph compression, a significant number of occurrences of {0,1} and {1,2} are replaced by R_1 and R_3, respectively, resulting in the original rule R_2 being replaced only once in the actual compression process. We refer to rules in the original graph that meet the specified frequency threshold but are replaced fewer times than the threshold during parallel compression as "fake rules". The presence of "fake rules" may negatively impact compression because storing the rules themselves also incurs a certain cost. Although during iterative compression, we cannot anticipate whether a certain rule will become a "fake rule" when counting its occurrences, we can obtain a simpler guarantee: In each iteration of the parallel rule-based subgraph compression module running in Laconic, the graph obtained can be smaller than the previous one. The fundamental idea behind Laconic's rule compression is to set a frequency threshold and only retain rules in the bucket that appear more frequently than this threshold, thereby ensuring the effectiveness of rule substitution.

In the following proof, we derive the "stable frequency threshold". This threshold guarantees that any parallel compression with a frequency threshold greater than the stable frequency threshold can result in a reduction of the graph size in each compression step.

Proof to validity. Calculating the contribution of a single rule to reducing redundancy is challenging, but we can express the space savings brought about by all rule replacements during an iteration of compression as: $\sum_{j\in\text{replace}} j \cdot single_replace - \sum_{i\in\text{freq}} i \cdot single_cost$. "$\sum_{j\in\text{replace}} j \cdot single_replace$" represents the space savings from removing all edges that have been replaced by rules, and "$\sum_{i\in\text{freq}} i \cdot single_cost$" represents the space overhead required to store all newly added rules.

Any captured binary rules are added as a new row to the adjacency list of the original graph. For an input in the CSR format, the cost of adding a rule vertex and two neighbors is 1 (from CSR_{vlist}) + 2 (from CSR_{elist}), which is 3. Since we use a replacement method similar to leftmost reduction when replacing rules, the replaced rules do not affect the rules before them. For example, replacing the neighbor pair {1,2} of a vertex with R_1 does not decrease the replacement count of {0,1}, because if {0,1} is recorded in all rules and also in the neighbor list of that vertex, {0,1} shall be replaced before {1,2}, which contradicts the replacement of {1,2}. However, it affects the rules behind it. Therefore, a one-time replacement of a rule can reduce the occurrence of other rules by at most one occurrence. This means that if the total occurrence count of a recorded rule is j, there will be at least $\frac{j}{2}$ rule replacements happening. Thus, the number of replacements is at least *(frequency threshold)×(number of rules) / 2*. We can set the threshold to at least 6 to ensure efficient compression, because in the case where the frequency threshold is set to 6, we can ensure that each rule can have an average of at least 6/2=3 effective replacements. As discussed previously, in the CSR format, the cost of adding a rule vertex and two adjacent edges is 3.

Correctness. In parallel rule-based compression, compared to serial compression, different threads detect different rules. Our goal is to prove that our algorithm does not affect the correctness of the compression results. That is, upon decompression, the graph is restored to its original state before compression.

Proof to correctness. We can abstract this problem as follows. In context-free grammar, all productions have only one non-terminal symbol on the left-hand side and two symbols on the right-hand side, regardless of whether they are terminals or non-terminals. Furthermore, each non-terminal symbol appears only once on the left-hand side of a production. Our goal is to demonstrate the uniqueness of leftmost reduction and leftmost derivation.

Uniqueness of leftmost reduction. We prove that for any state during the reduction process, the next production accepted for the leftmost reduction is unique. The proof is as follows. Suppose there are two different leftmost reductions, $A1$ and $A2$, for the neighbor array of a vertex. Let $k1$ and $k2$ be the first non-terminal symbols in $A1$ and $A2$, respectively, which are different. Since each production corresponds to consecutive occurrences of vertex pairs, there must exist a subscript such that $k1 < k2$ or $k2 < k1$, which contradicts

the definition of leftmost reduction. Therefore, the next production accepted for the leftmost reduction is unique.

Uniqueness of leftmost derivation. For leftmost derivation, since each non-terminal symbol appears only once on the left-hand side of all productions, there are no alternative results when converting non-terminal symbols into terminal symbols. This guarantees the uniqueness of the derivation result.

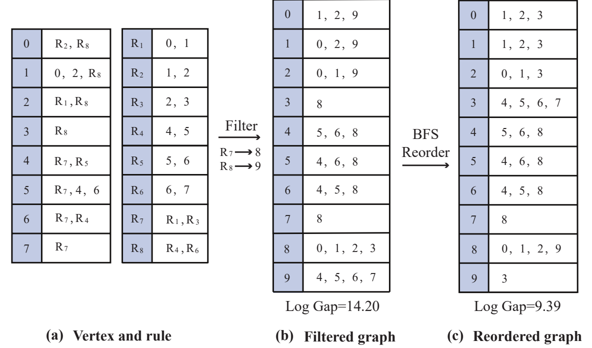
4.4 Graph Expression Recovery Module

4.4.1 ***Design.*** Unlike normal rule compression engines, Laconic has a two-layer compression module, so it is necessary to use the graph expression recovery module after rule compression to reduce the performance degradation caused by rule compression, as well as to enhance the localization of graphs and improve the overall performance. It comprises two main parts. First, the locality recovery sub-module initiates with infrequent rule filtering to eliminate rules that occur rarely during rule-based compression, caused by parallelism and cross-depth matching. This step effectively reduces redundancy introduced during compression, similar to prior work in the literature [112]. Second, the reordering process is employed to restore the original vertex arrangement disrupted by rule compression. For instance, rule compression may alter a vertex's original adjacency list from {1, 2, 3, 4} to {1, $|V|$+1, 4}, leading to a considerable increase in the "gaps" between neighboring vertices. To address this, we employ a BFS-based vertex reordering technique, effectively restoring the graph's locality. This sub-module exhibits low complexity and can be efficiently parallelized, thereby satisfying the low-latency requirements of graph computation systems.

The second part of the module is the parallel encoding compression sub-module, which is responsible for compressing the reordered graph using encoding techniques. Since in graph computation processes, we usually extract all neighbors of a vertex rather than individual neighbor, we can choose encoding methods with low amortized decompression overhead rather than low random decompression overhead. In this case, we choose Delta Encoding [86], which is a lightweight encoding method suitable for ordered data. It has good compression performance and provides $O(1)$ amortized decompression complexity per vertex. Moreover, it can be efficiently parallelized.

By incorporating the locality recovery submodule and parallel encoding compression submodule, we enhance the overall compression rate of Laconic, reducing both the redundancy in the graph and the computation time.

4.4.2 ***Example.*** We illustrate the specific processes of rule filtering and locality recovery in Figure 5. During the filtering phase in Figure 5 (b), we traverse all the neighbors of the entire adjacency list and differentiate between the frequent rules R_7 and R_8, and the infrequent rules R_1 to R_6 based on their frequencies. Infrequent rules are replaced with the element pairs they represent (e.g., replacing R_1 with 0,1 in the neighbor of vertex 2). Next, we use BFS reordering to reorder the vertex IDs and the neighbors within each vertex in the filtered graph. We define the sum logarithm of the difference between each neighbor and its previous neighbor in the adjacency list of vertex i as its "log gap". This is because the effective bit cost of encoding each vertex's neighbors in delta



(a) Vertex and rule (b) Filtered graph (c) Reordered graph

Figure 5: Example on locality recovery sub-module.

encoding is the logarithmic sum of the neighbor gaps in that vertex. The "log gap" of all vertices in the graph can be expressed as $\sum_{i\ in\ vertex} \sum_{j\ in\ i.neighbor} \log_2(j - j.prev_neighbor)$. After rule compression is completed, some vertices' "log gap" will significantly increase due to the newly introduced rule vertices, and reordering effectively addresses this issue. In Figure 5 (c), we observe that after reordering, the graph's log gap is reduced by 33%.

5 EVALUATION

5.1 Experimental Setup

Evaluation methodology. We conduct a comparative evaluation between Laconic and five state-of-the-art graph processing and analytics solutions, CompressGraph [20], Ligra [91], Ligra+ [92], WebGraph [5, 10], and GridGraph [120]. The rationale for selecting these competitors lies in the following aspects. First, we consider whether the competitors share the same motivation as Laconic. For example, Ligra+ [92] also reduces memory usage by introducing a lightweight compression module while efficiently analyzing graphs. Similarly, GridGraph [120], like Laconic, guarantees to perform graph analytics within restricted memory. Second, we choose competitors that excel in graph compression and graph analytics. WebGraph [5, 10] is renowned for its exceptional compression ratios, while CompressGraph [20] achieves efficient fast-reading analysis of compressed graphs. Third, these competitors have been discussed in many previous works [17, 30, 82, 95, 97, 101, 102, 104, 110]. The performance evaluation primarily focuses on four aspects: peak memory usage throughout the compression and computation process, time and space overhead during computation, compression ratio, and compression time.

Benchmarks. We evaluate five graph applications, including connected components (CC), betweenness centrality (BC), triangle counting (Triangle), single-source shortest path (SSSP), and PageRank (PR). These benchmarks have been widely used in previous studies [48, 68, 78, 117, 119, 120].

Datasets. We evaluate Laconic using the graphs listed in Table 1. $|V|$ represents the number of vertices, and $|E|$ represents the number of edges. These graphs are obtained from WebGraph [9, 10] and the Stanford Network Analysis Project (SNAP) [60], which have

been extensively utilized in prior studies [55, 77, 78, 117, 119]. The graphs are originally stored in an adjacency list format.

Table 1: Datasets.

Dataset	Graph	Abbreviation	$\|V\|$	$\|E\|$	Size
1	web-BerkStan	BERKSTAN	685,231	7,600,581	63MB
2	in-2004	IN	1,382,909	16,917,053	153MB
3	eu-2005	EU@2005	862,665	19,235,139	153MB
4	uk-2007-05@1000000	UK@1000000	1,000,000	41,247,159	322MB
5	indochina-2004	INDOCHINA	7,414,866	194,109,311	1.50GB
6	arabic-2005	ARABIC	22,744,080	639,999,458	4.93GB
7	it-2004	IT	41,291,594	1,150,725,436	8.88GB
8	gsh-2015-host	GSH	68,660,142	1,802,747,600	13.94GB
9	sk-2005	SK	50,636,154	1,949,412,601	14.9GB
10	uk-2007-05	UK	105,896,555	3,738,733,648	28.65GB
11	clueweb12	CLUE	978,408,098	42,574,107,469	324.49GB
12	uk-2014	UK@2014	787,801,471	47,614,527,250	360.63GB

Platform. We conduct a standard performance evaluation of Laconic on a CPU server equipped with an Intel Core i7-12700K CPU, featuring 12 cores and 20 threads. The server has 128GB of global memory, and the operating system used is Ubuntu 20.04.01. To demonstrate Laconic's superiority in handling large graphs such as *uk-2014* and *clueweb12*, we also measure Laconic's performance on another server with an Intel Xeon Gold 6230 CPU @ 2.10GHz, featuring 40 cores and 80 threads. The server has 600GB global memory, and the operating system used is Ubuntu 20.04.6.

5.2 Peak Memory Usage Throughout Compression and Computation Processes

We conduct a comparative analysis of Laconic against various state-of-the-art solutions, including traditional rule-based compression (CompressGraph), and encoding-based compression algorithms (Ligra and Ligra+). In this part, our comparison focuses on peak memory usage during the compression process, as the maximum memory consumption of compression significantly exceeds that of the subsequent graph analytics process. We also evaluate the peak memory usage of Ligra as a benchmark for direct graph processing. Moreover, to verify Laconic's scalability with large graphs, we involve two large datasets *clueweb12* and *uk-2014* on a machine with 600GB memory, using a batch size of 100 under high memory pressure. The results are summarized in Table 2.

Table 2: Peak memory usage during graph compression and processing.

Dataset	Original Size	CompressGraph	Ligra	Ligra+	Laconic
UK@2014	360.63 GB	-	-	-	**261.65 GB**
CLUE	324.49 GB	-	-	-	**288.76 GB**
UK	28.65 GB	65.376 GB	154.90 GB	121.85 GB	**24.86 GB**
SK	14.90 GB	57.40 GB	80.10 GB	62.87 GB	**13.89 GB**
GSH	13.94 GB	84.45 GB	74.80 GB	58.85 GB	**29.48 GB**
IT	8.88 GB	34.77 GB	47.59 GB	37.42 GB	**9.68 GB**
ARABIC	4.93 GB	19.16 GB	26.11 GB	20.46 GB	**5.24 GB**
INDOCHINA	1.50 GB	6.27 GB	7.71 GB	6.00 GB	**2.42 GB**
UK@1000000	322 MB	966.71 MB	1.54 GB	1.18 GB	**375.88 MB**
EU@2005	153 MB	744.83 MB	750.30 MB	577.27 MB	**281.91 MB**
IN	153 MB	834.35 MB	693.31 MB	540.61 MB	**314.00 MB**
BERKSTAN	63MB	393.63MB	250.96MB	230.62MB	**100.31 MB**

We have the following findings. First, Laconic's average peak memory usage during graph analytics is 1.32× the size of the original graph. This ratio increases to 1.43× for 10 batches and decreases to 0.81× for 100 batches. Second, CompressGraph, Ligra+, and Ligra exhibit peak memory usage of 4.37×, 3.60×, and 4.53× over the original graph size, respectively. Our solution reduces 70% memory usage on average. Third, on the *uk-2014* dataset, Laconic demonstrates lower peak memory usage than the original graph, with a peak memory of only 73.5% of the original graph size throughout processing. This reduction can be attributed to both the larger number of compression batches chosen for *uk-2014* and the good locality inherent in the dataset. Furthermore, the peak memory demands of Ligra+ and CompressGraph surpass the available system memory, preventing us from completing their compression on the *clueweb12* and *uk-2014* datasets. Consequently, these datasets will be excluded from the subsequent experiments to ensure a fair and valid comparison.

5.3 Batch Configuration

As discussed in Section 4.2, when dividing subgraphs for batch compression, choosing an appropriate subgraph size is crucial. If the subgraph size is too small, rule-based compression may deteriorate, leading to larger subgraphs for the subsequent encoding-based compression, which increases encoding compression time and peak memory usage. On the other hand, too many subgraphs can raise the peak memory usage of rule-based compression. Thus, finding a balance is essential to reduce rule-based compression while ensuring low peak memory usage for encoding-based compression, thereby enhancing Laconic's graph processing capabilities under resource constrained environment.

As shown in Figure 6, after conducting experiments on real-world graphs, we have the following findings. First, setting the batch size to 10 effectively eliminates the high peak memory usage caused by large original graphs during the rule-based compression, addressing the bottleneck of reducing peak memory in the compression process. Second, we do not recommend setting the batch size greater than 10, as for most graphs, further increasing the batch size has a limited impact on reducing peak memory; excessively large batch sizes can disrupt the locality of the original graph and degrade compression performance. Third, when the batch size is less than 20, the peak memory overhead of encoding-based compression in Laconic does not exceed that of rule-based compression. Because the initial step of rule-based compression significantly reduces the graph size by several magnitudes, the encoded graph ends up being much smaller than the original graph.

5.4 Compression Ratio

We conduct a comprehensive comparison of compression ratios for Laconic, CompressGraph, and Ligra+ on different graphs. The original graph sizes stored using adjacency lists and the sizes of the compressed data obtained with these techniques are provided. The compression ratios for each method (Laconic, CompressGraph, and Ligra+) are reported, representing the ratio of the original data size to the respective compressed data size. Table 3 presents a summary of our experimental results.

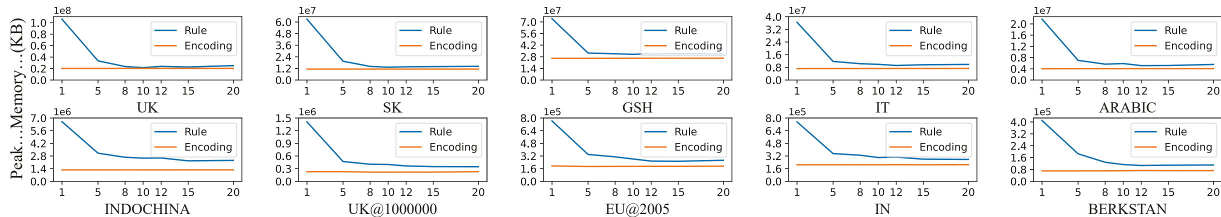


Figure 6: The relationship between peak memory and the number of batches.

Table 3: Comparison of compression ratios and compressed graph sizes of Ligra+, CompressGraph and Laconic.

Dataset	Original Size	Ligra+ Compressed Size	Ligra+ Compression Ratio	CompressGraph Compressed Size	CompressGraph Compression Ratio	Laconic Compressed Size	Laconic Compression Ratio
UK@2014	360.63GB	-	-	-	-	23.89GB	16.30
CLUE	324.49GB	-	-	-	-	38.99GB	8.32
UK	28.65GB	5.44GB	5.26	4.48GB	6.38	3.48GB	8.23
SK	14.90GB	5.04GB	2.95	2.62GB	5.68	1.85GB	8.05
GSH	13.94GB	5.93GB	2.35	6.47GB	2.15	4.76GB	2.93
IT	8.88GB	3.04GB	2.92	1.72GB	5.16	1.22GB	7.29
ARABIC	4.93GB	1.71GB	2.88	0.96GB	5.12	699MB	7.22
INDOCHINA	1.50GB	525MB	2.92	295MB	5.20	216MB	7.10
UK@1000000	322MB	106MB	3.03	47MB	6.76	32MB	9.88
EU@2005	153MB	61MB	2.50	43MB	3.52	32MB	4.69
IN	153MB	58MB	2.63	48MB	3.16	35MB	4.32
BERKSTAN	63MB	30MB	2.10	26MB	2.43	20MB	3.15

We have the following observations. First, Laconic achieves the highest average compression ratio of 7.29, making it 2.47× more efficient in compression than Ligra+ (with a ratio of 2.95) and 1.58× more efficient than CompressGraph (with a ratio of 4.61). This highlights Laconic's effectiveness in reducing redundancy in graphs with low overhead. However, Ligra+ and CompressGraph incur relatively high auxiliary memory overhead during the compression process, making them impractical for compressing large graphs, such as the *clueweb12* and *uk-2014* datasets, under memory constrained environments. Second, Laconic demonstrates the highest compression ratio on the *uk-2014* dataset. This can be attributed to the dataset's vertices having a substantial number of neighbors (an average of 60.44 neighbors per vertex) and exhibiting good locality, which allows Laconic to fully exploit the advantages of multi-dimensional redundancy elimination in the dataset.

5.5 Compression Speedup

Encoding-based compression leverages lightweight encoding techniques, such as Delta [86] and RLE [1], with well-established parallelization implementations, yielding an amortized time complexity of $O(1)$ per edge. In contrast, the time complexity of rule-based compression remains a significant challenge, as no parallel rule-based compression method has been proposed so far. As a result, rule-based compression constitutes the most time-consuming part of the Laconic compression process. To address this issue, we introduce a novel parallel rule capture and replacement algorithm in Section 4.3. This novel approach effectively reduces the compression time, addressing the high time complexity associated with rule-based compression and enhancing the efficiency of the overall Laconic compression process.

We compare the compression time of Laconic's parallel rule-based compression, CompressGraph compression, and Ligra+ compression. To ensure each compression effectively reduces the size of the graph, we set the frequency threshold from 4 to 6, as discussed in Section 4.3.4. Our evaluation entails measuring the parallel rule-based compression, serial rule-based compression, and Ligra+ compression with frequent thresholds set to 4 and 5, respectively, and measuring the compression time.

The results are presented in Table 4. We have the following findings. First, Laconic's parallel rule-based compression achieves a 9.64× speedup compared to Ligra+ in each round of iteration, and a 13.92× speedup compared to the serial rule-based (CompressGraph) compression. Even when considering multiple iterations and capturing deeper-level rules, Laconic's parallel rule-based compression still achieves favorable results, with an average speedup of 3.16× compared to Ligra+ in the sum of the first three rounds of iterations, and an average speedup of 4.52× compared to the serial rule-based (CompressGraph) compression. Second, in most cases, the compression time of the parallel version of rule-based compression decreases as the number of iterations increases. This trend can be attributed to the reduced graph size resulting from previous compression, along with a decrease in the number of rules that can be captured and replaced in subsequent rounds. These factors collectively reduce the computational workload. Third, the rule-based compression time generally increases with the growth of the data size. However, the specific compression time exhibits significant fluctuations, contingent on the dataset's characteristics. For example, during parallel compression, the compression time of the *gsh-2015-host* dataset approaches or even surpasses that of the serial compression of the *uk-2007-05* dataset.

Table 4: Compression time evaluation. PFT is short for *parallel frequency threshold*.

Dataset	Ligra+ (s)	CompressGraph (s)	Laconic (s) - PFT = 5				Laconic (s) - PFT = 4			
			Iteration 1	Iteration 2	Iteration 3	Total	Iteration 1	Iteration 2	Iteration 3	Total
UK	605.03	690.43	81.67	66.82	42.13	190.62	82.51	67.42	44.58	194.51
SK	326.81	334.71	23.61	22.14	22.43	68.19	24.51	23.41	21.94	69.87
GSH	324.03	812.63	61.70	62.88	59.44	183.03	61.17	61.66	56.46	179.31
IT	175.25	198.43	15.56	14.55	13.96	44.08	15.40	13.98	12.76	42.14
ARABIC	53.54	105.34	5.08	3.62	2.88	11.58	5.11	2.80	3.78	11.70
INDOCHINA	28.50	51.66	7.62	7.08	7.02	22.73	7.62	7.79	6.70	22.12
UK@1000000	3.38	4.07	0.37	0.33	0.29	1.00	0.40	0.31	0.32	1.03
EU@2005	1.57	2.39	0.22	0.21	0.21	0.66	0.24	0.22	0.24	0.70
IN	1.25	2.23	0.19	0.18	0.16	0.54	0.20	0.18	0.19	0.58
BERKSTAN	0.97	1.51	0.11	0.10	0.10	0.31	0.12	0.11	0.09	0.32

Unlike systems that directly analyze the original graph, a graph analytics system with a compression module can store intermediate results and subsequently read the compressed results for graph analytics, reducing the subsequent I/O time. In this case, the peak memory usage throughout the process depends only on the space required for analyzing the compressed graph. To simulate this scenario, we store the intermediate results compressed by Ligra+, CompressGraph, and Laconic, and conduct several tasks on them.

5.6 Peak Memory and Time Consumption During Computation

We assess the peak memory usage and time consumption of the system when performing analytics tasks on the compressed graphs using Ligra+, CompressGraph, and Laconic compression, as shown in Figure 7 and Figure 8. We find that although the peak memory overhead of encoding-based and rule-based compression methods during graph analytics is significantly lower than the memory overhead during compression, the results obtained solely from rule-based compression or encoding-based compression do not guarantee satisfactory performance in both time and space during computation. Rule-based compression achieves an average speedup of 1.47× compared to encoding-based compression during computation, but the peak memory usage when performing computation tasks on rule-based compressed graphs is 2.15× higher than that of encoding-based compressed graphs. Therefore, in addition to the rule-based and encoding-based compression utilized in Laconic, we introduce the locality recovery sub-module in Section 4.4. During the execution of these modules, we observe that the Laconic-compressed graphs achieve an average peak memory usage reduction of 27% compared to Ligra+, and 66% compared to CompressGraph. Regarding the time usage for graph analytics, the Laconic-compressed graphs demonstrate speedups of 2.13× and 1.47× compared to Ligra+ and CompressGraph, respectively.

5.7 Additional Baselines and Discussion

Various graph systems have emerged, such as disk-oriented [35, 56, 71, 77] and non-computation-oriented compression solutions [10, 12, 28, 29]. We compare Laconic with these prominent systems.

Comparison with non-computation-oriented compression. The field of compression has seen significant advancements, with notable contributions such as GZIP [28] and LZW [29]. In the realm of graph compression, WebGraph [10] stands out as a prime example. In detail, we conduct a thorough comparison between Laconic and WebGraph, considering both compression ratios and the efficiency of operations on the compressed graphs. To assess the space cost, we examine the ratio of the compressed graph's size to the number of edges, referred to as bits per edge (*bpe*). The performance on the compressed graph is analyzed in terms of both time and space. For our evaluation, we primarily focus on CC across all datasets, as it effectively traverses each vertex of the graph. Other workloads exhibit similar performance characteristics. In our experiments, WebGraph achieves a *bpe* of 3.04 across the datasets, while Laconic achieves a slightly higher *bpe* of 4.17. It is worth noting that despite Laconic's compressed graphs having a higher compression ratio, it processes at a remarkable rate of 1.78×10^7 edges per second, which is impressively 11.7× faster than WebGraph.

Comparison with the disk-based graph system. Disk-based graph systems store the entire graph on disk and handle large graphs by partitioning them and storing a portion in memory. GridGraph [120] is an exemplary approach for querying large graphs with limited memory resources. It partitions extensive graph data into blocks, each containing a subset of the graph's vertices and edges. By reducing the size of the processed graph at any given time, it aims to control memory usage. We conduct a performance comparison between Laconic and GridGraph on CC and PR. To guarantee the same amount of available memory, we set the available memory during GridGraph execution to match the peak memory of the compressed graph after Laconic compression. Additionally, based on the results in GridGraph, we set the number of blocks to 64 (8×8) to ensure maximum efficiency with minimal partitioning overhead. Our findings reveal that, on average, Laconic outperforms GridGraph by 74× on PR and 23× on CC. This disparity arises because GridGraph necessitates repeated block I/Os during PR execution, which is considerably slower than memory access.

6 RELATED WORK

Compression has been widely applied in various fields [19, 42, 57, 58, 61–66, 87, 109]. In this section, we show the related work of graph compression from different dimensions.

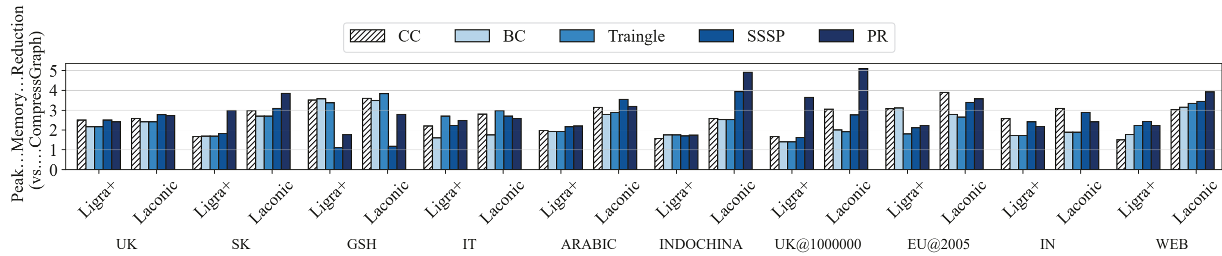


Figure 7: Peak memory reduction.

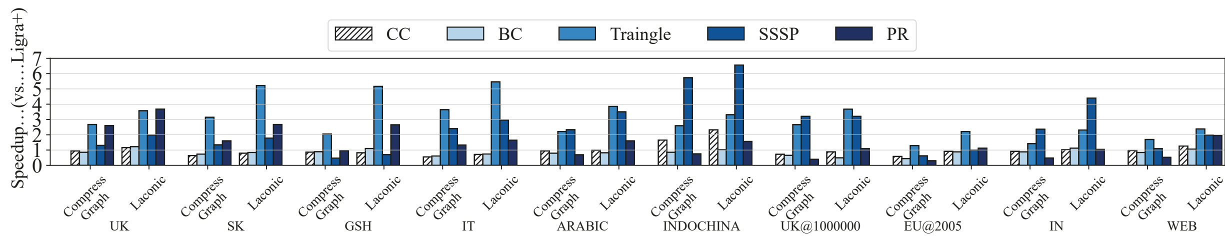


Figure 8: Performance speedup.

Memory reduction in graph analytics. In graph analysis systems, peak memory often limits the maximum graph size that a system can handle in a given environment. Representative works on reducing memory usage during graph analytics include memory graph computing [92] and distributed graph computing systems [119], both focusing on minimizing peak memory consumption. Graph partitioning techniques are widely used in disk-based graph systems [21, 35, 56, 71, 77, 120], allowing the storage of the entire graph on disk and a portion of it in memory to handle large graphs. For example, Fan et al. [35] proposed a hierarchical contraction scheme for querying large graphs. Such techniques can also accelerate graph computation [18, 33]. In Laconic, we employ batch compression to reduce memory consumption during rule compression and minimize peak memory utilization.

Compressed data direct processing. Many works attempt to ensure the effectiveness of graph queries while compressing the graph [3, 34, 53, 74, 75]. Some direct processing methods on compressed data have been proposed, involving advanced techniques based on trees, indexes, and suffix arrays [8, 24, 37–40, 45, 50, 81]. Succinct [2] is a representative method for queries on compressed data. Many graph compression technologies and systems support access to vertex neighbors without complete decompression [10, 91]. However, in practical graph applications, graph redundancy is not fully utilized to improve performance. Researchers have found that data reuse can be achieved through data redundancy utilization, leading to improved program performance. For example, TADOC [112–116] is a representative rule-based compression method that enables efficient text analytics without decompression. These works demonstrate the potential application of rule-based compression. The study [25] proposed applying rule-based compression methods from text to graph data. Previous works also have utilized grammar compression on strings [11, 14, 23, 26, 43, 44, 108, 117] and graphs [20, 72, 73]. However, the high compression overhead of many graph computing systems with compression modules results in compression times far exceeding the analytics time.

7 CONCLUSION

This paper presents Laconic, a fast and efficient method for graph compression under resource constraints. By employing data segmentation, rule compression, and encoding compression, Laconic achieves an average peak memory reduction of 70% during compression and 66% during computation. Moreover, Laconic significantly reduces rule compression time by 93% compared to traditional methods, resulting in a 2.47× higher compression ratio and a 2.12× performance speedup. The paper demonstrates the ability of Laconic to capture redundancies at different graph levels and proposes tailored compression methods for various redundancies. Additionally, a parallel rule-based compression method is introduced to expedite the compression process. Experimental results validate the promising practicality of Laconic.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China (62322213, 62172419, 62122090, and 62072461), Beijing Nova Program (20230484397 and 20220484137), Beijing Natural Science Foundation (L222006), and the Research Funds of Renmin University of China. Q. Xu, F. Zhang, Z. Chen, J. Guan, J. Fan, Y. Zhang, and X. Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and the School of Information, Renmin University of China. Feng Zhang is the corresponding author of this paper.

REFERENCES

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 671–682.

[2] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling queries on compressed data. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 337–350.

[3] Sebastian E Ahnert. 2013. Power graph compression reveals dominant relationships in genetic transcription networks. *Molecular BioSystems* 9, 11 (2013), 2681–2685.

[4] Peter A Alsberg. 1975. Space and time savings through large data base compression and dynamic restructuring. *Proc. IEEE* 63, 8 (1975), 1114–1122.

[5] Alberto Apostolico and Guido Drovandi. 2009. Graph compression by BFS. *Algorithms* 2, 3 (2009), 1031–1044.

[6] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 781–792.

[7] André M Bastos and Jan-Mathijs Schoffelen. 2016. A tutorial review of functional connectivity analysis methods and their interpretational pitfalls. *Frontiers in systems neuroscience* 9 (2016), 175.

[8] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *the Journal of machine Learning research* 3 (2003), 993–1022.

[9] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[10] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*. 595–602.

[11] Dhruba Borthakur et al. 2008. HDFS architecture guide. *Hadoop Apache Project* 53, 1-13 (2008), 2.

[12] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. 2009. k 2-trees for compact web graph representation. In *International symposium on string processing and information retrieval*. Springer, 18–30.

[13] Gregory Buehrer and Kumar Chellapilla. 2008. A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 international conference on web search and data mining*. 95–106.

[14] Michael Burrows and David Wheeler. 1994. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer.

[15] Venkat Venkat Bala Chandar. 2010. *Sparse graph codes for compression, sensing, and secrecy*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[16] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. 2005. The smallest grammar problem. *IEEE Transactions on Information Theory* 51, 7 (2005), 2554–2576.

[17] Hongtao Chen, Mingxing Zhang, Ke Yang, Kang Chen, Albert Zomaya, Yongwei Wu, and Xuehai Qian. 2023. Achieving Sub-second Pairwise Query over Evolving Graphs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3575693.3576173

[18] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Trans. Parallel Comput.* 5, 3 (2019). https://doi.org/10.1145/3298989

[19] Yixin Chen, Guozhu Dong, Jiawei Han, Jian Pei, Benjamin W Wah, and Jianyong Wang. 2006. Regression cubes with lossless compression and aggregation. *IEEE Transactions on Knowledge and Data Engineering* 18, 12 (2006), 1585–1599.

[20] Zheng Chen, Feng Zhang, JiaWei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. 2023. CompressGraph: Efficient Parallel Graph Analytics with Rule-Based Compression. *Proc. ACM Manag. Data* 1, 1 (2023), 31.

[21] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. Nxgraph: An efficient graph processing system on a single machine. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 409–420.

[22] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 219–228.

[23] Trishul M Chilimbi. 2001. Efficient representations and abstractions for quantifying and exploiting data reference locality. *ACM SIGPLAN Notices* 36, 5 (2001), 191–202.

[24] Trishul M Chilimbi and Martin Hirzel. 2002. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 199–209.

[25] Francisco Claude and Gonzalo Navarro. 2010. Fast and compact web graph representations. *ACM Transactions on the Web (TWEB)* 4, 4 (2010), 1–31.

[26] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[27] François Deliège and Torben Bach Pedersen. 2010. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proceedings of the 13th international conference on Extending Database Technology*. 228–239.

[28] Peter Deutsch et al. 1996. GZIP file format specification version 4.3. (1996).

[29] HN Dheemanth. 2014. LZW data compression. *American Journal of Engineering Research* 3, 2 (2014), 22–26.

[30] Laxman Dhulipala, Jakub Łącki, Jason Lee, and Vahab Mirrokni. 2023. TeraHAC: Hierarchical Agglomerative Clustering of Trillion-Edge Graphs. *Proc. ACM Manag. Data* 1, 3, Article 221 (nov 2023), 27 pages. https://doi.org/10.1145/3617341

[31] Kasper Dinkla, Michel A Westenberg, and Jarke J van Wijk. 2012. Compressed adjacency matrices: Untangling gene regulatory networks. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (2012), 2457–2466.

[32] Peter Elias. 1975. Universal codeword sets and representations of the integers. *IEEE transactions on information theory* 21, 2 (1975), 194–203.

[33] Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Xiaojian Luo, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2020. Application Driven Graph Partitioning. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1765–1779. https://doi.org/10.1145/3318464.3389745

[34] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. 2012. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 157–168.

[35] Wenfei Fan, Yuanhao Li, Muyang Liu, and Can Lu. 2022. A Hierarchical Contraction Scheme for Querying Big Graphs. In *Proceedings of the 2022 International Conference on Management of Data*. 1726–1740.

[36] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *The world wide web conference*. 417–426.

[37] Andrea Farruggia, Paolo Ferragina, and Rossano Venturini. 2014. Bicriteria data compression: Efficient and usable. In *European Symposium on Algorithms*. Springer, 406–417.

[38] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. 2009. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)* 13 (2009), 1–12.

[39] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *Journal of the ACM (JACM)* 52, 4 (2005), 552–581.

[40] Paolo Ferragina, Igor Nitto, and Rossano Venturini. 2009. On the bit-complexity of Lempel-Ziv compression. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 768–777.

[41] Linton C Freeman, Douglas Roeder, and Robert R Mulholland. 1979. Centrality in social networks: II. Experimental results. *Social networks* 2, 2 (1979), 119–141.

[42] Shangqian Gao, Feihu Huang, Jian Pei, and Heng Huang. 2020. Discrete model compression with resource constraint for deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1899–1908.

[43] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms*. Springer, 326–337.

[44] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 17–30.

[45] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2004. When indexing equals compression: experiments with compressing suffix arrays and applications.. In *SODA*, Vol. 4. 636–645.

[46] Ankit Gupta and Sergio Verdú. 2009. Nonlinear sparse-graph codes for lossy compression. *IEEE Transactions on Information Theory* 55, 5 (2009), 1961–1975.

[47] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 233–245.

[48] Wentao Han, Xiaowei Zhu, Ziyan Zhu, Wenguang Chen, Weimin Zheng, and Jianguo Lu. 2016. A comparative analysis on Weibo and Twitter. *Tsinghua Science and Technology* 21, 1 (2016), 1–16.

[49] Cecilia Hernández and Gonzalo Navarro. 2011. Compression of web and social graphs supporting neighbor and community queries. In *Proc. 5th ACM Workshop on Social Network Mining and Analysis (SNA-KDD). ACM*.

[50] Wing-Kai Hon, Tak Wah Lam, Wing-Kin Sung, Wai-Leuk Tse, Chi-Kwong Wong, and Siu-Ming Yiu. 2004. Practical aspects of Compressed Suffix Arrays and FM-Index in Searching DNA Sequences.. In *ALENEX/ANALC*. Citeseer, 31–38.

[51] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

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

[53] Xiaowei Jiang, Xiang Zhang, Feifei Gao, Chunan Pu, and Peng Wang. 2013. Graph compression strategies for instance-focused semantic mining. In *China Semantic Web Symposium and Web Science Conference*. Springer, 50–61.

[54] Jon M Kleinberg. 1999. Hubs, authorities, and communities. *ACM computing surveys (CSUR)* 31, 4es (1999), 5–es.

[55] Christine Klymko, David Gleich, and Tamara G Kolda. 2014. Using triangles to improve community detection in directed networks. *arXiv preprint arXiv:1404.5874* (2014).

[56] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. {GraphChi}:{Large-Scale} Graph Computation on Just a {PC}. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 31–46.

[57] Laks VS Lakshmanan, Jian Pei, and Yan Zhao. 2003. Efficacious data cube exploration by semantic summarization and compression. In *Proceedings 2003 VLDB Conference*. Elsevier, 1125–1128.

[58] Laks VS Lakshmanan, Jian Pei, and Yan Zhao. 2003. Socqet: semantic olap with compressed cube and summarization. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 658–658.

[59] N.J. Larsson and A. Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732. https://doi.org/10.1109/5.892708

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

[61] Jinbao Li and Jianzhong Li. 2005. Data sampling control and compression in sensor networks. In *International Conference on Mobile Ad-Hoc and Sensor Networks*. Springer, 42–51.

[62] Jinbao Li and Jianzhong Li. 2007. Data sampling control, compression and query in sensor networks. *International Journal of Sensor Networks* 2, 1-2 (2007), 53–61.

[63] Jianzhong Li, Qianqian Ren, et al. 2011. Compressing information of target tracking in wireless sensor networks. *Wireless Sensor Network* 3, 02 (2011), 73.

[64] Jianzhong Li, Doron Rotem, and Jaideep Srivastava. 1999. Aggregation algorithms for very large compressed data warehouses. In *VLDB*, Vol. 99. 651–662.

[65] JZ Li, Doron Rotem, and Harry KT Wong. 1987. A new compression method with fast searching on large databases. (1987).

[66] Jianzhong Li and Jaideep Srivastava. 2002. Efficient aggregation algorithms for compressed data warehouses. *IEEE Transactions on Knowledge and Data Engineering* 14, 3 (2002), 515–529.

[67] Yongquan LIANG, Qiuyu SONG, Zhongying Zhao, Hui Zhou, and Maoguo Gong. 2023. BA-GNN: Behavior-aware graph neural network for session-based recommendation. *Frontiers of Computer Science* 17, 6, Article 176613 (2023), 0 pages. https://doi.org/10.1007/s11704-022-2324-x

[68] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, et al. 2018. Shentu: processing multi-trillion edge graphs on millions of cores in seconds. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 706–716.

[69] Wei Liu, Andrey Kan, Jeffrey Chan, James Bailey, Christopher Leckie, Jian Pei, and Ramamohanarao Kotagiri. 2012. On compressing weighted time-evolving graphs. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 2319–2322.

[70] István Lukovits. 2000. A compact form of the adjacency matrix. *Journal of chemical information and computer sciences* 40, 5 (2000), 1147–1150.

[71] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*. 527–543.

[72] Sebastian Maneth and Fabian Peternek. 2016. Compressing graphs by grammars. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 109–120.

[73] Sebastian Maneth and Fabian Peternek. 2018. Grammar-based graph compression. *Information Systems* 76 (2018), 19–45.

[74] Hossein Maserrat and Jian Pei. 2010. Neighbor query friendly compression of social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 533–542.

[75] Hossein Maserrat and Jian Pei. 2012. Community preserving lossy compression of social networks. In *2012 IEEE 12th International Conference on Data Mining*. IEEE, 509–518.

[76] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

[77] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! But at what {COST}?. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

[78] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: understanding graph computing in the context of industrial solutions. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[79] Craig G Nevill-Manning and Ian H Witten. 1997. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research* 7 (1997), 67–82.

[80] Craig G Nevill-Manning and Ian H Witten. 1997. Linear-time, incremental hierarchy inference for compression. In *Proceedings DCC'97. Data Compression Conference*. IEEE, 3–11.

[81] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. 1996. *Pthreads programming: A POXIS standard for better multiprocessing*. Vol. 19. O'reilly Sebastopol, CA, USA.

[82] Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal C. Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. 2023. Accelerating Sparse Data Orchestration via Dynamic Reflexive Tiling. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (, Vancouver, BC, Canada,) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/3582016.3582064

[83] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano Indexes. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*. 273–282.

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

[85] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D Owens. 2017. Multi-GPU graph analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 479–490.

[86] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Michael A Kozuch, Phillip B Gibbons, and Todd C Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *2012 21st international conference on parallel architectures and compilation techniques (PACT)*. IEEE, 377–388.

[87] Qianqian Ren, Jianzhong Li, and Jinbao Li. 2007. An efficient clustering-based method for data gathering and compressing in sensor networks. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, Vol. 1. IEEE, 823–828.

[88] Ryan A Rossi and Rong Zhou. 2018. Graphzip: a clique-based sparse graph compression method. *Journal of Big Data* 5, 1 (2018), 1–14.

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

[90] Quan Shi, Yanghua Xiao, Nik Bessis, Yiqi Lu, Yaoliang Chen, and Richard Hill. 2012. Optimizing K2 trees: A case for validating the maturity of network of practices. *Computers & Mathematics with Applications* 63, 2 (2012), 427–436.

[91] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[92] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *2015 Data Compression Conference*. IEEE, 403–412.

[93] Harmanjit Singh and Richa Sharma. 2012. Role of adjacency matrix & adjacency list in graph theory. *International Journal of Computers & Technology* 3, 1 (2012), 179–183.

[94] Jie Sun, Erik M Bollt, and Daniel Ben-Avraham. 2008. Graph compression—save information by exploiting redundancy. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 06 (2008), P06001.

[95] Hongshi Tan, Xinyu Chen, Yao Chen, Bingsheng He, and Weng-Fai Wong. 2023. LightRW: FPGA Accelerated Graph Dynamic Random Walks. *Proc. ACM Manag. Data* 1, 1, Article 90 (may 2023), 27 pages. https://doi.org/10.1145/3588944

[96] Nan Tang, Qing Chen, and Prasenjit Mitra. 2016. Graph stream summarization: From big bang to big crunch. In *Proceedings of the 2016 International Conference on Management of Data*. 1481–1496.

[97] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and Efficient Full-Graph GNN Training for Large Graphs. *Proc. ACM Manag. Data* 1, 2, Article 143 (jun 2023), 23 pages. https://doi.org/10.1145/3589288

[98] Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson. 2017. MILC: Inverted list compression in memory. *Proceedings of the VLDB Endowment* 10, 8 (2017), 853–864.

[99] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 993–1008.

[100] Jiajia Wang, Weizhong Zhao, Xinhui Tu, and Tingting He. 2023. A novel dense retrieval framework for long document retrieval. *Frontiers of Computer Science* 17, 4, Article 174609 (2023), 0 pages. https://doi.org/10.1007/s11704-022-2041-5

[101] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2023. HongTu: Scalable Full-Graph GNN Training on Multiple GPUs. *Proc. ACM Manag. Data* 1, 4, Article 246 (dec 2023), 27 pages. https://doi.org/10.1145/3626733

[102] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2023. HongTu: Scalable Full-Graph GNN Training on Multiple GPUs. *Proc. ACM Manag. Data* 1, 4, Article 246 (dec 2023), 27 pages. https://doi.org/10.1145/3626733

[103] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.

[104] Zhibin Wang, Longbin Lai, Yixue Liu, Bing Shui, Chen Tian, and Sheng Zhong. 2023. I/O-Efficient Butterfly Counting at Scale. *Proc. ACM Manag. Data* 1, 1, Article 34 (may 2023), 27 pages. https://doi.org/10.1145/3588714

[105] Zhigang Wang, Ning Wang, Jie Nie, Zhiqiang Wei, Yu Gu, and Ge Yu. 2023. A lock-free approach to parallelizing personalized PageRank computations on GPU. *Frontiers of Computer Science* 17, 1, Article 171602 (2023), 171602 pages. https://doi.org/10.1007/s11704-022-1546-2

[106] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442.

[107] Wikipedia Contributors. 2023. Sparse matrix. https://en.wikipedia.org/wiki/Sparse_matrix. Accessed: 2023.

[108] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First international workshop on graph data management experiences and systems*. 1–6.

[109] Chi Yang, Xuyun Zhang, Changmin Zhong, Chang Liu, Jian Pei, Kotagiri Ramamohanarao, and Jinjun Chen. 2014. A spatiotemporal compression based approach for efficient big data processing on cloud. *J. Comput. System Sci.* 80, 8 (2014), 1563–1583.

[110] Feng Yao, Qian Tao, Wenyuan Yu, Yanfeng Zhang, Shufeng Gong, Qiange Wang, Ge Yu, and Jingren Zhou. 2024. RAGraph: A Region-Aware Framework for Geo-Distributed Graph Processing. *Proc. VLDB Endow.* 17, 3 (jan 2024), 264–277. https://doi.org/10.14778/3632093.3632094

[111] Chang Ye, Yuchen Li, Bingsheng He, Zhao Li, and Jianling Sun. 2023. Large-Scale Graph Label Propagation on GPUs. *IEEE Transactions on Knowledge and Data Engineering* (2023).

[112] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1522–1535.

[113] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Zwift: A programming framework for high performance text analytics on compressed data. In *Proceedings of the 2018 International Conference on Supercomputing*. 195–206.

[114] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2020. Enabling efficient random access to hierarchically-compressed data. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1069–1080.

[115] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2022. POCLib: A high-performance framework for enabling near orthogonal processing on compression. *IEEE transactions on Parallel and Distributed Systems* 33, 2 (2022), 459–475.

[116] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30, 2 (2021), 163–188.

[117] Feng Zhang, Jidong Zhai, Bo Wu, Bingsheng He, Wenguang Chen, and Xiaoyong Du. 2019. Automatic irregularity-aware fine-grained workload partitioning on integrated architectures. *IEEE Transactions on Knowledge and Data Engineering* (2019).

[118] Amelie Chi Zhou, Juanyun Luo, Ruibo Qiu, Haobin Tan, Bingsheng He, and Rui Mao. 2022. Adaptive Partitioning for Large-Scale Graph Analytics in Geo-Distributed Data Centers. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2818–2830.

[119] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 301–316.

[120] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*. 375–386.