CompressGraph: Efficient Parallel Graph Analytics with Rule-Based Compression

ZHENG CHEN, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China
FENG ZHANG[*], Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China
JIAWEI GUAN, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China
JIDONG ZHAI, Department of Computer Science and Technology, Tsinghua University, China
XIPENG SHEN, Computer Science Department, North Carolina State University, United States
HUANCHEN ZHANG, Tsinghua University, China and Shanghai Qi Zhi Institute, China
WENTONG SHU, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China
XIAOYONG DU, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China

Modern graphs exert colossal time and space pressure on graph analytics applications. In 2022, Facebook social graph reaches 2.91 billion users with trillions of edges. Many compression algorithms have been developed to support direct processing on compressed graphs to address this challenge. However, previous graph compression algorithms do not focus on leveraging redundancy in repeated neighbor sequences, so they do not save the amount of computation for graph analytics. We develop CompressGraph, an efficient rule-based graph analytics engine that leverages data redundancy in graphs to achieve both performance boost and space reduction for common graph applications. CompressGraph has three advantages over previous works. First, the rule-based abstraction of CompressGraph supports the reuse of intermediate results during graph traversal, thus saving time. Second, CompressGraph has intense expressiveness to support a wide range of graph applications. Third, CompressGraph scales well under high parallelism because the context-free rules have few dependencies. Experiments show that CompressGraph provides significant performance and space benefits on both CPUs and GPUs. On evaluating six typical graph applications, CompressGraph can achieve

[*] Feng Zhang is the corresponding author of this paper.

Authors' addresses: Zheng Chen, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China, chenzheng123@ruc.edu.cn; Feng Zhang, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China, fengzhang@ruc.edu.cn; Jiawei Guan, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China, guanjw@ruc.edu.cn; Jidong Zhai, Department of Computer Science and Technology, Tsinghua University, China, zhaijidong@tsinghua.edu.cn; Xipeng Shen, Computer Science Department, North Carolina State University, United States, xshen5@ncsu.edu; Huanchen Zhang, Tsinghua University, China and Shanghai Qi Zhi Institute, China, huanchen@tsinghua.edu.cn; Wentong Shu, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China, 2019201418@ruc.edu.cn; Xiaoyong Du, Key laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, China, duyong@ruc.edu.cn.

2836-6573/2023/5-ART4 $15.00
https://doi.org/10.1145/3588684

1.97× speedup on the CPU, while 3.95× speedup on the GPU, compared to the state-of-the-art CPU and GPU methods, respectively. Moreover, CompressGraph can save an average of 71.27% memory savings on CPU and 70.36 on GPU.

CCS Concepts: • **Information systems → Graph-based database models**; • **Theory of computation →** *Graph algorithms analysis.*

Additional Key Words and Phrases: graph analytic, compression, compressed data direct processing

ACM Reference Format:
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, Article 4 (May 2023), 31 pages. https://doi.org/10.1145/3588684

1 INTRODUCTION

Graph analytics has become the basis in many fields, such as social networks [3, 6, 10, 45, 79, 134, 156], machine learning [45, 48, 103, 108, 110, 122, 127, 128, 148, 149], and data mining [19, 20, 23, 31, 36, 45, 66, 82, 123, 130, 150]. However, the tremendous growth of graph sizes poses great pressure in graph analytics systems. For example, as of 2022, the Facebook social graph involves 2.91 billion users and trillions of edges [93]. Such magnitude of graphs brings challenges from both time and space perspectives, making the traditional adjacency matrix approach impractical [34]. Large graph representations are particularly unfriendly to heterogeneous accelerators such as GPUs, because these accelerators usually use discrete memory that has limited capacity and is connected through PCIe.

There is an extensive literature on graph representations [9, 12, 13, 16, 17, 49, 116, 117, 119], among which the adjacency list [119] is the most widespread. The adjacency list for a graph node is simply a list of its neighboring vertices, and is space-efficient when the graph is sparse. However, the size of the adjacency lists still grows linearly with the number of edges in the graph, which means that graph analytics tasks still need to process the same number of edges as they would for the adjacency matrix, making this approach suboptimal in handling the explosion of graph sizes.

Fortunately, graphs in the real world such as web graphs [13] and social graphs [132] contain a lot of redundancies. Our experiments (detailed in Section 6) show that 74.6% of the edges are repeatedly stored when adjacency list is used to represent real web and social graphs. Previous lossless compression methods achieve significant space savings by leveraging such redundancies [9, 12, 13, 16, 49, 116, 117]. However, applications operating on these compressed graphs have to decode on-the-fly, causing performance degradation. Moreover, the irregular access patterns and dependencies of the decoding procedures make it difficult to parallelize.

We propose to use rule-based compression to facilitate efficient parallel graph analytics. There are three main advantages. First, rule-based compression can effectively reduce redundancies in a graph, where a repeated sequence of neighboring vertices is represented by a rule that takes less space. For example, as shown in Figure 1, when we perform PageRank [104] and connectivity analysis [8] to find "small worlds" on a large social network graph, we find that 57.8% of people have mutual friends, showing a great potential for data reuse. Second, rule-based compression can reduce redundant computations by caching and reusing the process results for rules. Prior works show that similar computation reusing techniques achieve 2.2× speedup for 77.5% data redundancy on data science applications [142, 146]. Finally, our rule-based compression allows processing directly on compressed graph, thus avoiding the expensive graph decoding operations. Additionally, rule-based computing has been proved to be GPU friendly [140], and the compression format effectively expands the processing capacity of GPUs on graph analytics.

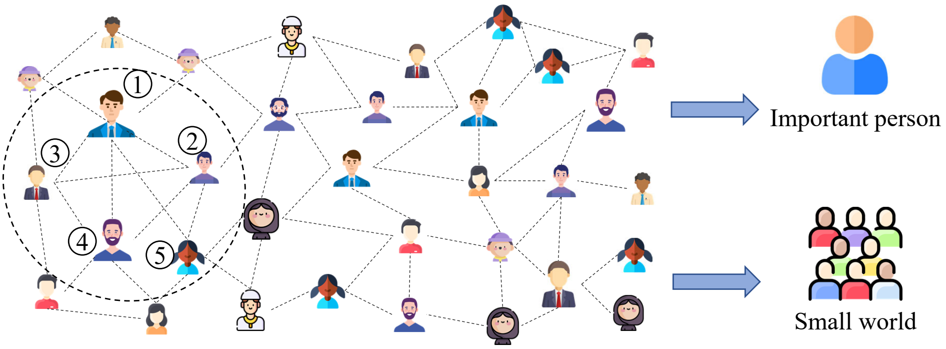


Fig. 1. Use case of social network analytics. Users 3, 4, and 5 are repeated neighbors of both users 1 and 2.

Performing efficient graph processing directly on compressed graphs requires handling the following three challenges. The first challenge is to achieve space and computation reductions simultaneously on a graph. The second challenge is to enable the compression format to serve a wide range of graph applications efficiently. Finally, the proposed algorithm must be highly parallel so that it can be accelerated by GPUs to achieve even better performance.

In this paper, we present CompressGraph, our graph analytics engine leveraging rule-based compression. CompressGraph represents graphs using a hierarchy of rules to achieve efficient compression and computation reuse. The engine supports a wide range of graph applications, thanks to our built-in finite state machine (FSM) that has a high expressive capacity. The FSM also has fewer dependencies compared to the systems that require decoding (detailed in Section 6.2) to allow higher parallelism. We developed a synchronization-free parallel strategy in CompressGraph, which has both CPU and GPU supports.

We compared CompressGraph against state-of-the-art solutions (Ligra+ [116, 117] on CPU, and Gunrock [129] on GPU) on six common graph algorithms: breadth-first search (BFS), single-source shortest path (SSSP), connected component (CC), pagerank (PR), topological sorting (TP), and Hyperlink-Induced Topic Search (HITS). Experiments show that CompressGraph accelerates the applications by 1.97× when executing on CPUs. For GPUs, the speedup reaches 3.95×. Meanwhile, CompressGraph is space-efficient, with a 74.6% compression rate ($1 - compressed_{size}/raw_{size}$), and consumes 71.27% and 70.36% less memory for CPU and GPU processing, respectively.

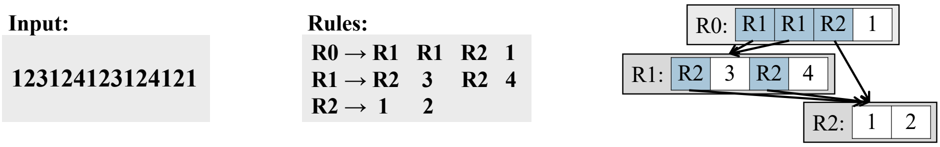
We summarize the contributions of this work as follows.

- We exhibit our insights in the rule-based graph compression, and point out that we can perform compressed graph direct processing via rule interpretation.
- We develop CompressGraph, a graph analytics engine that can perform efficient compressed graph direct processing on both CPU and GPU.
- We evaluate CompressGraph with six typical graph analytics applications, and demonstrate its benefits over the state-of-the-art graph processing methods.

2 BACKGROUND AND MOTIVATION

2.1 Existing Graph Compression

Researchers have studied compressed graph representations extensively [5, 21, 54, 113]. The most commonly used formats are adjacency matrices and adjacency lists. Compared to adjacency matrices, adjacency lists are more compact because graphs are usually sparse in practice. Accordingly, many



(a) Original data. (b) TADOC compressed data. (c) DAG representation.

Fig. 2. A rule-based compression.

graph compression studies [12, 13, 16, 117] investigate encoding the adjacency list to maximize space savings.

Graph encoding. Encoding the neighbors of vertices can significantly reduce storage space. We list four representative graph compression techniques as discussed in the study [9]: 1) *Variable-length encoding* employs variable-length codes to represent the vertices in the adjacency list instead of fixed byte representation such as *int* and *long*. 2) *Reference encoding* begins by extracting the same neighbors of two vertices and then uses reference coding to encode them. 3) *Interval encoding* stores consecutive vertices (e.g. *x,x+1,...,x+k*) using interval boundaries *x* and *x+k*. 4) *Gap encoding* preserves the difference between neighbors $N_1(v)$-v, $N_2(v)$-$N_1(v)$, ..., $N_{d_v}(v)$-$N_{d_v-1}(v)$, rather than the vertex ID $N_i(v)s$ themselves, given a neighbor sequence "$N_1(v)$, $N_2(v)$, ..., $N_{d_v}(v)$" of vertex v. All of these encoding-based approaches must spend extra cycles decoding the graph on the fly when performing graph analytics.

GPU processing. In recent years, heterogeneous accelerators, represented by GPUs, are becoming more and more popular, and have already been used in graph systems [50, 55, 59, 63, 106, 129, 152]. However, GPUs have limited capacity and on-the-fly decompression is difficult to parallelize, making existing graph compression solutions unsuitable for GPU parallelism.

2.2 Redundancies in Real-World Graphs

We define the "redundancy ratio" of a graph as the portion of the adjacency list where a neighbor sequence (with length $\geq$ 2) appears more than once. Redundancies exist in a wide range of graph applications [8, 39, 44, 64, 104, 133]. Here, we examine two real-world graphs from different categories (see Section 6.1 for graph details). *sk-2005* is a representative web graph where algorithms such as page rank [104] can be applied to mine hubs and authorities [64]. *hollywood-2009* is a social network graph where the vertices represent actors/actresses and the edges indicate cooperation in the same movie. Our analysis shows that the redundancy ratios for *sk-2005* and *hollywood* are as high as 85.3% and 57.8%, respectively. Such a high fraction of repetition motivates us to develop a rule-based compression scheme to improve the efficiency of graph analytics.

2.3 Rule-Based Compression

Rule-based compression has been proved to be a big success in data science [29, 100, 101, 142–144, 146]. One representative is TADOC (text analytics directly on compression) [142–144, 146], which utilizes a set of context-free grammar rules to represent text data. TADOC uses a rule to denote a repeated piece of content, and a rule can contain subrules and pieces of original data. TADOC can further use a DAG to organize the rules, and thus can transform the original text analytics tasks into a DAG traversal problem.

Data compression. In rule-based compression, a rule represents a subsequence that occurs multiple times in the original data. The compressor reads a string of length N sequentially and stores digrams in a hash table (a digram is a pair of two adjacent elements). Each time a new element is scanned, the compressor checks the hash table for its digram. The compressor substitutes a

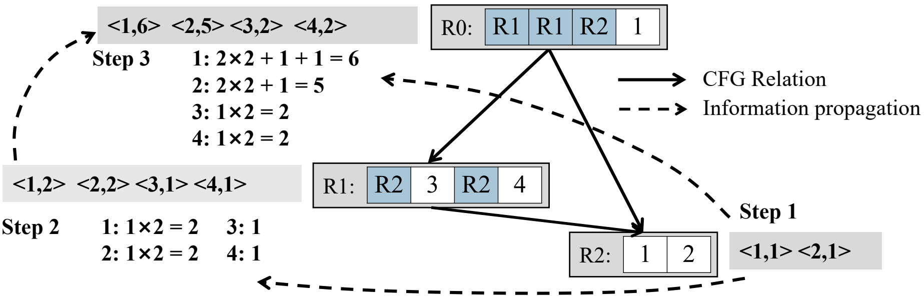


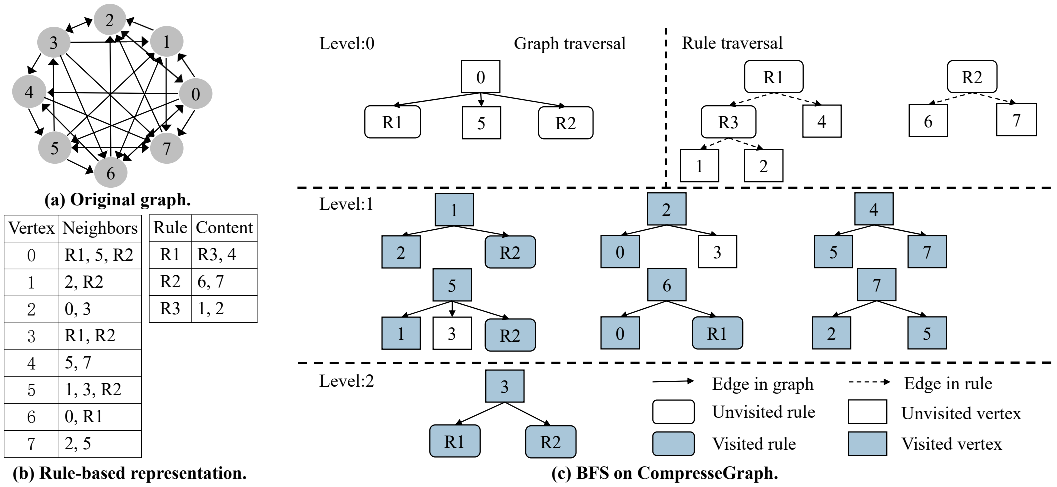
Fig. 3. An example of using TADOC for counting word frequencies.

digram with a rule if it already exists; otherwise, it creates a new digram record. After replacing a digram with a new rule, the compressed format contains fewer elements. Then, the compressor restores unique subrules. Note that the compressor scans the data only once and the hash table digram search complexity is O(1). Each substitution shortens the string by one, so the maximum number of substitutions is N. Accordingly, the compression has O(N) complexity.

We illustrate the process in Figure 2. The numbers in Figure 2 (a) represent different elements, which are compressed into different rules denoted by R_i in Figure 2 (b). Specifically, when we scan up to {1,2,3,1,2}, the compressor substitutes “1,2” with rule R1, and generates {R1,3,R1}. Similarly, the compressor would transform the scan sequence {1,2,3,1,2,4,1,2,3} to {R1,3,R1,4,R1,3}. Because there are repeated occurrences of “R1,3”, it is replaced with a new rule “R2”. This process continues until we obtain string {R4,R4,R1,1} after the full scan, where R4 ⟶ “R3,4”, R3 ⟶ “R1,R2”, R2 ⟶ “R1,3”, and R1 ⟶ “1,2”. Because R2 and R3 appear only once, the compressor restores R2 and R3, making R4 ⟶ {R1,3,R1,4}. After renumbering the rules, the original string is represented in the form shown in Figure 2 (b). In addition, the rules can be organized as a DAG, as shown in Figure 2 (c).

Application. Figure 3 shows an application where the task of counting word frequencies is transformed to a graph traversal. Each node in the DAG performs a local word count for its rule and passes the result to the parent node. Because we compute only once for each rule (although it appears multiple times in the hierarchy), we save duplicate computations. Specifically, instead of scanning the entire sequence of uncompressed data (Figure 2 (a)), we first obtain word frequencies {<1,1>, <2,1>} for R2. We then perform the task in R1 and get an intermediate result {<R2,2>, <3,1>, <4,1>}. After substituting R2 with its prior computation result, the final word frequencies for R1 become {<1,2>, <2,2>, <3,1>, <4,1>}. We repeat the process for R0 and deliver the final result. Note that we perform the word count only once for R1 and R2, despite that the sequences are repeated multiple times in the original data.

Opportunities. We argue that the rule-based compression also fits graph analytics, where repeated neighbor sequences can be represented using rules. Figure 4 (b) shows an example where the neighbor sequence of each vertex is represented by a combination of vertices and recursive rules. However, a naive application of TADOC on the adjacency list cannot support efficient processing on the compressed graph directly. One reason is that graph algorithms are often iterative, causing multiple traversals of the DAG of rules. In the next section, we describe the conceptual framework of our system that makes this idea practical.



(a) Original graph.

Vertex	Neighbors
0	R1, 5, R2
1	2, R2
2	0, 3
3	R1, R2
4	5, 7
5	1, 3, R2
6	0, R1
7	2, 5

Rule	Content
R1	R3, 4
R2	6, 7
R3	1, 2

(b) Rule-based representation.

(c) BFS on CompresseGraph.

Fig. 4. An example of BFS using CompressGraph.

3 CONCEPTUAL FRAMEWORK

We develop a graph analytics engine called CompressGraph, which can perform graph analytics tasks without decompression. In this section, we introduce the conceptual framework of CompressGraph. We first describe our basic idea of conducting graph processing directly on CompressGraph. Then, we model graph traversal on the compressed graph using a finite state machine to generalize our solution to a wide range of graph applications.

3.1 Basic Idea of CompressGraph

The core idea behind CompressGraph is data reuse. To reuse data, we calculate only once for the results of multiple repeated computations in graph traversal and then reuse the duplicated intermediate results in subsequent computations. In graph application, we can attach special attributes to the rules, such as whether certain rules of neighbors have been traversed or not. If a rule has been processed by one parent, its other parents do not need to further scan the rule including its subrules, thus saving computation.

Design. CompressGraph adopts a two-level traversal design, including a graph traversal (i.e., processes vertices without rules) and a rule traversal. The two-level traversal process contains four steps. 1) *Initialization.* CompressGraph puts the initial vertex into a vertex queue. 2) *Graph traversal.* CompressGraph selects its neighbor vertices that have not been visited, marks them as visited, and puts them into the vertex queue. 3) *Rule traversal.* CompressGraph recursively expands the rules it encounters into vertex sequences using the rule graph. It also marks the not-yet-visited vertices and puts them into the vertex queue. 4) CompressGraph pops out the first element from the vertex queue and repeats the procedure from the second step.

Example. We use breadth-first search (BFS) to illustrate the above process in Figure 4. We start with the *graph traversal* step for vertex 0 whose neighbors include $\{R1, 5, R2\}$. We mark vertex 5 as visited and put it into the queue for next iterations. Then, we deal with $\{R1, R2\}$ in the *rule traversal* step, during which we recursively expand $R1$ by marking and visiting $\{R1, R3, 1, 2, 4\}$, and recursively expand $R2$ by marking and visiting $\{R2, 6, 7\}$. The vertex queue now includes $\{5, 1, 2, 4, 6, 7\}$. Note that during the rule traversal of $R1$, we mark $R3$ as visited to avoid potential re-traversals later. Figure 4 (c) shows the remaining BFS process. Because $\{R1, R2, R3, 1, 2, 4, 5, 6, 7\}$ are already marked as visited during the first iteration, we can skip them in later ones.

3.2 Graph Traversal Modeling

To generalize our idea, we use a finite state machine (FSM) [71] to model the graph traversal process. We first introduce a programming model to describe the compressed graph applications. Second, we use a FSM to describe the state transition in graph traversal. Third, we use an example to show the graph traversal process.

Programming model. CompressGraph functions in a vertex-centric mechanism. In each iteration, CompressGraph is driven by changes in vertices. Both CompressGraph and other graph processing frameworks [63, 96, 129] that use vertex-centric models include two major steps: (1) *Advance*. Given an input vertex set, the system visits the neighbors of the set and generates the set of neighbors that satisfy the condition as the output. (2) *Compute*. The compute step performs the operation on all the neighbors of the set of vertices being processed. These two steps can be defined as *condition* and *operation*, where *condition* corresponds to the *advance* step and *operation* corresponds to the *compute* step. The input of the condition is a vertex u, and the output is a Boolean value. The input of the operation is a vertex pair $<u, v>$, where u belongs to the currently processed vertex set, and v is a neighbor of u. A similar idea has also been adopted by Gunrock [129].

The difference between CompressGraph and other vertex-centric graph analytics frameworks [63, 116, 129] is that CompressGraph directly processes the compressed graph, which contains two different types of elements: vertices and rules. We treat rules as a new type of vertices. Therefore, CompressGraph comprises two types of vertices: traditional vertices and rules. CompressGraph also includes four types of edges: *v2v*, *v2r*, *r2v*, and *r2r*, where v represents a vertex and r represents a rule. For different types of vertices and edges, their conditions and operations are also different. We propose the definition for CompressGraph based on the vertex-centric programming model.

Through examination of various graph applications, we develop a unified conceptual framework, which can be represented by a six-element tuple <*Graph*, *Operation*, *Condition*, *Result*, $State_{start}$, $State_{end}$>. The conceptual framework can capture common graph traversal operations. The detailed definition is as follows.

- *Graph*, which represents the compressed graph, denoted as <V, R, E>, where V represents the set of vertices, and R represents the set of rules. E represents the set of edges in the graph. An edge can be represented by <src, dst>, which has four different types: <v,v>, <v,r> , <r,v>, and <r,r>, where v is an element of V and r is an element of R. $Graph$ satisfies the condition: $E \subseteq (V \cup R) \times (V \cup R)$.
- *Operation*, which has four types in CompressGraph, corresponds to four types of edges. We use O_{v2v}, O_{v2r}, O_{r2v}, and O_{r2r} to indicate the operations between vertices and rules.
- *Condition*, which determines whether an element needs to be processed in the next iteration, including C_v for vertices and C_r for rules. In each iteration, for a vertex or rule, the *Condition* function checks all the neighbors. Then, CompressGraph adds the neighbors that satisfy the condition function to the set of elements to be processed in the next iteration. In detail, given a vertex v or rule r, *Condition* can check all their properties, including the user-defined *Result* field and common properties such as degree we provide. For example, in topological sorting (TP), *Condition* checks whether the in-degree of input element is 0.
- *Result*, which represents the data structure for the final result, and can be used in *Operation* and *Condition*.
- $State_{start}$, which represents the start state.
- $State_{end}$, which represents the end state.

Finite state machine for compressed graph traversal. We use (W, G, B) to represent the states of CompressGraph in graph traversal, where W denotes the set of elements (part of $V \cup R$) that

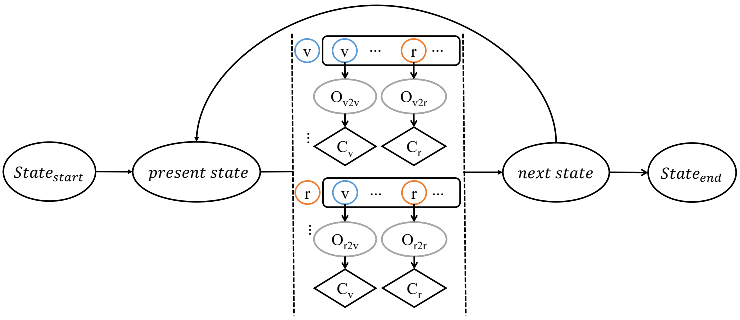


Fig. 5. Finite state machine in CompressGraph.

do not need to be processed, G denotes the set of elements that need to be processed, and B denotes the set of elements that have already been processed. We use a fixed-length array to maintain the state of each vertex and rule in the graph. All the vertices and rules are independent of each other without a nested relationship, so the space complexity of FSM is $O(V + R)$. Since there are only three states for a vertex or rule, we can represent the state using only two bits. Experiments show that such data structures occupy less than 2% of the total space, which is negligible. Both $State_{start}$ and $State_{end}$ are special states of (W, G, B). Therefore, the states in CompressGraph satisfy the conditions in Equation 1.

$$\begin{aligned} &W, G, B \subseteq (V \cup R), \\ &W \cup G \cup B = V \cup R, \\ &(W \cap G) \cup (W \cap B) \cup (G \cap B) = \varnothing \end{aligned} \tag{1}$$

Then, the graph traversal in CompressGraph can be described as a finite state machine, as shown in Figure 5. The finite state machine starts from $State_{start}$, and ends at $State_{end}$. We formally define the state transition as operating on all elements belonging to G (part of $V \cup R$). For an element, v or r, in G, CompressGraph traverses all its neighbors and executes the operation based on the type of edges (O_{v2v}, O_{v2r}, O_{r2v}, and O_{r2r}). Afterwards, CompressGraph checks all its neighbors to see whether the condition, C_v or C_r, is met. If so, CompressGraph adds the neighbor to G, which needs to be processed in the next iteration. When CompressGraph reaches $State_{end}$, CompressGraph stops.

Correctness proof. Assume a vertex v in the original graph, whose neighbor set is $\{u_1, u_2, ..., u_n\}$. For a neighbor u_i, if edge $<v, u_i>$ is in the compressed graph, we can process the edge $<v, u_i>$ with O_{v2v} and process u_i with C_v to determine whether to add it to $State.G$. If edge $<v, u_i>$ is not in the compressed graph, then there must be a path $<v, r_1, ..., r_m, u_i>$ between v and u_i in the compressed graph with only rules. In this case, we perform rule traversal to process the edge $<v, u_i>$ in the original graph. We use O_{v2r} to process edge $<v, r_1>$, O_{r2r} to process $<r_i, r_{i+1}>$, and O_{r2v} to process $<r_m, u_i>$. Besides, we use C_r to determine whether to add $\{r_1, r_2, ..., r_m\}$ to $State.G$. During rule traversal, we can use the *Result* field of rules to store the intermediate results. Thus, the programming model and finite state machine of CompressGraph can handle any vertex and its neighbors with the rule-based compression format.

Listing 1. BFS definition in CompressGraph.

```
 1  CompressGraph = {Graph; Operation; Condition; Result, State_start, State_end};
 2  Graph = {V, R, E};
 3  class Operation{
 4      void Ov2v(vertex src, vertex dst){
 5          if(dst.distance == INIT) { dst.distance = src.distance+1; }
 6      }
 7      void Ov2r(vertex src, rule dst){
 8          if(dst.distance == INIT) { dst.distance = src.distance+1; }
 9      }
10      void Or2v(rule src, vertex dst){
11          if(dst.distance == INIT) { dst.distance = src.distance; }
12      }
13      void Or2r(rule src, rule dst){
14          if(dst.distance == INIT) { dst.distance = src.distance; }
15      }
16  };
17  class Condition{
18      bool Cv(vertex V) { return V.distance == INIT; }
19      bool Cr(rule R) { return R.distance == INIT; }
20  };
21  class Result{
22      int distance;
23      Result(Graph G){
24          distance = INIT;
25      }
26  };
27  State_start = {V&R-{root}, {root}, null};
28  State_end = {U1, null, U2};
29  State_cur = State_start;
```

Example. We use BFS as an example to illustrate the graph processing using CompressGraph in Listing 1. The result of BFS is the distance from the root to all vertices (Line 22). The distances of all elements are initially set to $INIT$ (Lines 23 to 25). The O_{v2v} and O_{v2r} operations assign $distance$+1 of src to the $distance$ of dst if $dst.distance$ is equal to $INIT$ (Lines 4 to 9). The O_{r2v} and O_{r2r} operations set the distance value of dst to that of src if $dst.distance$ is equal to $INIT$ (Lines 10 to 15). The conditions of C_v and C_r represent whether the distance from the elements to the root has been updated (Lines 17 to 20). In BFS, $State_{start}$ is $< U, \{root\}, \varnothing >$, where U is $\{(V \cup R) - \{root\}\}$ (Line 27). $State_{end}$ is $< U1, \varnothing, U2 >$, where $U1 \cup U2 = V \cup R$ (Line 28). In this definition, because the distance of a rule and the distance of a vertex are changed once the rule (vertex) has been visited, they are put into G at most once. Therefore, we only need to process the rule once, after which all elements that access the rule do not need to retraverse the rule recursively again, thus achieving data reuse.

Figure 6 illustrates the state transition process of the BFS example in Section 3.1. In $State_{start}$, only vertex 0 is in G, and all the other elements are in W. In each state transition, we take out an element (v or r) in G, traverse all its neighbors, put the neighbors that have not been traversed from W into G, and put the element into B, to form the next state. The execution terminates when G is empty.

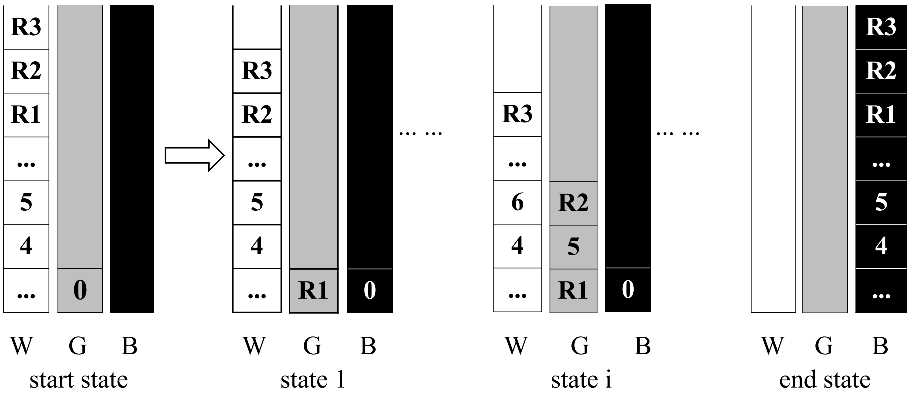


Fig. 6. Illustration of the state transition process. The white grid cells W represent the elements that do not need to be processed. The gray grid cells G represent the elements that need to be processed. The black grid cells B represent the elements that have been processed.

4 SYSTEM DESIGN

4.1 Overview

We show the overview of CompressGraph in Figure 7. After receiving the user input <*Graph*, *Operation*, *Condition*, *Result*, $State_{start}$, $State_{end}$>, CompressGraph engine starts to process the input, which consists of two main phases: an initialization phase and a two-level traversal phase (Section 4.2). The initialization phase is used to pre-scan the user input and initialize data structures. The two-level traversal phase is used to traverse the compressed graphs. GPU can accelerate the two-level traversal phase. To enable high parallelism in processing compressed graphs, we develop an inter-level synchronization-free graph traversal optimization (Section 4.3) and in-edge support to solve write conflict issues (Section 4.4) on GPU.

Novelties. We summarize the novelties of CompressGraph as follows. First, CompressGraph can improve performance and reduce storage space at the same time by means of data reuse in the rule-based compressed graph. Second, CompressGraph uses a finite state machine (FSM) innovatively to support a wide range of graph applications by parsing rules. Third, we propose a novel inter-level synchronization-free graph traversal and in-edge support optimization, which removes the last barrier to enable CompressGraph to take advantage of parallelism in both multi-core CPUs and GPUs.

4.2 CompressGraph Engine

This section introduces CompressGraph, which consists of two phases, initialization and two-level traversal, as shown in Figure 7.

Initialization. In the initialization phase, CompressGraph first checks whether $State_{start}$ and $State_{end}$ are valid. Second, CompressGraph loads the graph data, constructs necessary data structures, and builds functions for user-defined input. Third, CompressGraph conducts branch reduction since different types of operations can include the same content.

1) State checking. The rationality of the state determines whether CompressGraph can execute successfully. Wrong $State_{start}$ and $State_{end}$ can incur program errors, causing the program to crash. $State_{start}$ and $State_{end}$ must satisfy the constraints in Equation 1.

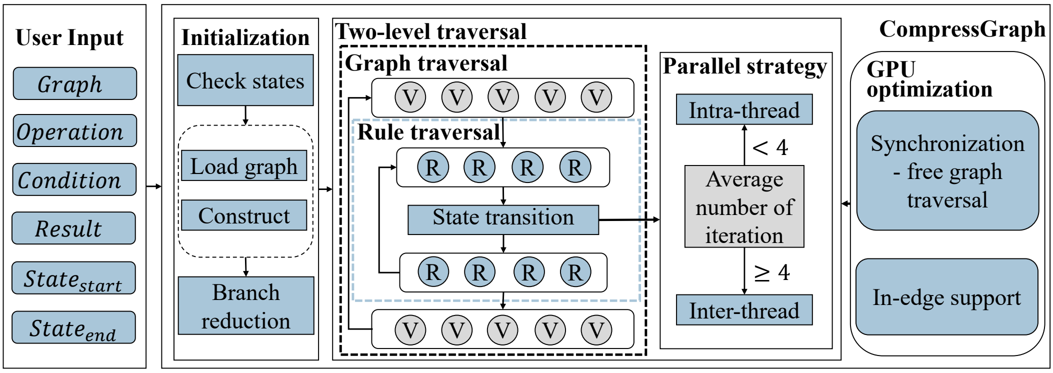


Fig. 7. CompressGraph overview.

2) Graph loading and construction. The elements of compressed graphs can be categorized into graph vertices and rules. CompressGraph loads the compressed graph into memory and records vertices and rules, respectively. Afterwards, the application-related data structures and operations are constructed.

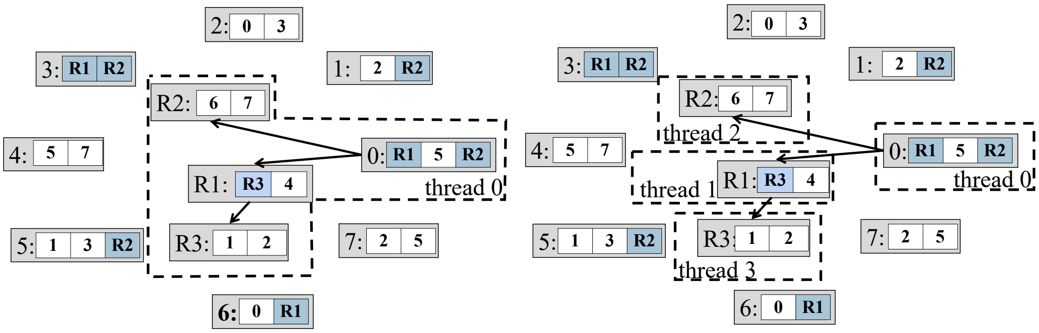
3) Branch reduction. We find that different types of operations can include the same content. For example, for BFS, O_{v2v} and O_{v2r}, O_{r2v} and O_{r2r}, and C_v and C_r are the same. These operations can be merged to reduce program branches for efficiency.

Two-level traversal. CompressGraph uses a two-level traversal model for graph applications, which includes graph traversal and rule traversal. The graph traversal serves to traverse vertices, whereas the rule traversal serves to traverse rules. The two-level traversal is a nesting of these two kinds of traversals. The outer layer is graph traversal, and the inner layer is rule traversal, as shown in Figure 7. Both graph traversal and rule traversal consist of iterations, and we use state transition to abstract the processing in each iteration of the two-level traversal. Because it is necessary to further traverse rules as vertex neighbors, each graph traversal iteration can include multiple rule traversal iterations. We define that the elements processed in each iteration in graph traversal and rule traversal are at the same level, and they can be processed in parallel. We further call each iteration in the graph traversal a *graph level*, because all elements in the same graph level belong to the same level in the uncompressed state.

Graph traversal. We call the traversal of the vertices graph traversal, which is the same as the uncompressed data. The input of each iteration in graph traversal is a set of vertices. After processing, a set of vertices is generated as the input of the next iteration. The CompressGraph engine guarantees that each iteration in the graph traversal obtains the same result as the uncompressed data, which naturally proves the correctness of our program.

Rule traversal. The rule traversal is nested in the graph traversal, where CompressGraph processes the DAG of rules. CompressGraph starts with a rule and traverses all its sub-rules recursively. For each rule encountered in the rule traversal process, CompressGraph decides whether to traverse downwards of the rule according to the user-defined condition C_r. If the rule has been traversed, which means that C_r is *false*, CompressGraph does not need to process the rule repeatedly, including its subrules, thus saving computation.

State transition. We use the state transition for graph traversal and rule traversal switching in the two-level traversal. We record only the start state and end state in one iteration to parallelize the state transition process without recording the intermediate states. Accordingly, the elements in a state transition can be processed in parallel. For example, there are several state transitions from



(a) Intra-thread rule traversal. **(b) Inter-thread rule traversal.**

Fig. 8. Different parallel strategies for rule traversal.

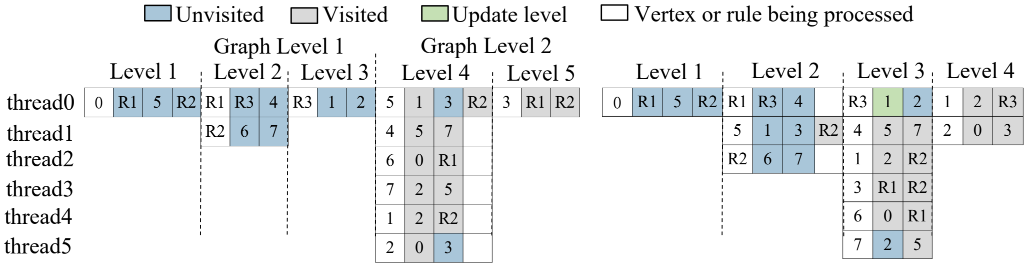
$state_i$ to $state_j$ in Figure 6. We record only the $state_i$ and $state_j$ because they are the beginning states of the two different traversal iterations. In detail, we can process {R2,5,R1} in parallel, and process {7,6,3,1,4,R3} in parallel.

We dynamically determine the parallel strategy for rule traversal in each state transition. There are two parallel strategies for rule traversal: (1) intra-thread rule traversal, and (2) inter-thread rule traversal, as shown in Figure 8. For intra-thread rule traversal in Figure 8 (a), CompressGraph creates N threads at the beginning of each graph traversal iteration, where N is the number of vertices in $State.G$. Rule traversal is processed recursively within the same thread. In contrast, inter-thread rule traversal, as shown in Figure 8 (b), processes rules in parallel in graph traversal, which creates a thread for each rule at the rule level. Inside the thread, the inter-thread rule traversal processes only the direct children of the rule. Intra-thread rule traversal requires fewer threads and synchronization times, but can cause excessive processing time for certain threads due to load imbalance problems. Inter-thread rule traversal can increase the degree of parallelism and avoid the long waiting time caused by the tail of long processing time for specific threads; however, it has more parallelism overhead than the intra-thread rule traversal. We calculate the average number of iterations needed to traverse each rule in $state.G$. If the number is higher than the threshold (four by default), we choose the inter-thread rule traversal; otherwise, we use the intra-thread rule traversal.

GPUs are widely used in graph analytics applications, and CompressGraph can also perform compressed graph direct processing on GPUs. In the rest part of this section, we introduce the optimizations of inter-level synchronization-free graph traversal (Section 4.3) and in-edge support for handling write conflicts (Section 4.4) to address the parallelism challenges of applying CompressGraph on GPU.

4.3 Inter-Level Synchronization-Free Graph Traversal

To handle the parallelism challenge of the complicated dependencies in the two-level traversal (graph-level and rule-level), we develop an inter-level synchronization-free graph traversal strategy. The basic idea is to avoid rule-level synchronization waiting at the end of the graph-level traversal to make full use of GPU capacity. In other words, our strategy enables rules and vertices at different graph levels to work simultaneously. Note that a corrective procedure is required for the parallel inconsistency issue.



(a) Inter-level synchronization-based graph traversal.

(b) Inter-level synchronization-free graph traversal.

Fig. 9. Difference between normal graph traversal and our proposed synchronization-free graph traversal.

General design. The general design of CompressGraph is as follows. The first step is to start from $State_{start}$ and process all the neighbors of the elements in the set of G. For vertices and rules, we put them into G according to the different conditions provided by the user. Second, for the vertices and rules that exist simultaneously in G, we directly process them in parallel without waiting for the end of a graph level. We call a parallel iteration a level. Third, CompressGraph repeats this process in a loop until it reaches $State_{end}$. Note that the vertices and rules in the same level can belong to different graph levels in this process. Fourth, simply using such synchronization-free traversal can incur inconsistency from the serial scheme, and therefore we develop a corrective procedure in the program to solve this issue.

Example. We use the graph data in Figure 4 as input, and show the difference between normal graph traversal and our proposed synchronization-free graph traversal in Figure 9. Figure 9 (a) shows the graph traversal with synchronization, which can be divided into different graph levels. A graph level contains different levels for rule traversal. The next graph level cannot start until CompressGraph reaches the end of the current graph level. For example, CompressGraph in graph level 1 processes all the neighbors of vertex 0, and ends only after the completion of all traversals of R1 and R2. Figure 9 (b) shows the synchronization-free graph traversal. With such optimization, traversals in different graph levels are overlapped, and CompressGraph can process as many elements as possible in each level. For example, at level 2, CompressGraph can process $R1$, $R2$, and vertex 5 at the same time.

For applications that need to obtain the distance from the root to the target vertex, CompressGraph requires an extra corrective procedure. In this example, using the operation defined in Section 3.1, the distance from the root to vertex 1 is 2. Because vertex 1 is the neighbor of vertex 5 when we first visit vertex 1, the distance from the root to vertex 5 is 1. However, the distance from the root to vertex 1 should be 1 because vertex 1 belongs to the neighbors of vertex 0. To solve this issue, we develop a corrective procedure. Every time we visit the vertex, we need to justify which of the original distance and the new distance is smaller; if the new distance is smaller, we need to update the original distance. In Figure 9 (b), we update the distance from the root to vertex 1 in level 3.

Algorithm. Algorithm 1 shows the pseudocode of our proposed synchronization-free graph traversal. This algorithm contains only one layer of loop for state transition, because there is no need to synchronize inside the loop to wait for the completeness of all rule operations. For each state transition, we process the vertices and rules in the candidate set in parallel. In a thread, the first step is to obtain the candidate element in Line 9. Next, we traverse all the neighbors of the element, and divide them into four categories for different operations and into two categories for different conditions from Lines 10 to 16. The loop stops when it reaches the end state. Algorithm 1 is a synchronization-free algorithm. In the model of two-level traversal mentioned in Section 4.2, CompressGraph waits for the end of the rule-level traversal in each iteration of the graph-level traversal. In detail, all threads are synchronized once in an iteration of graph-level traversal, waiting for the rule-level traversal to complete. In contrast, Algorithm 1 puts the graph-level and rule-level traversals into a single kernel shown in Lines 10-16, and allows different threads to process vertices and rules simultaneously. For example, Lines 11-12 represent the processing of vertices, while Lines 13-14 represent the processing of rules. In this way, we avoid synchronization between graph-level traversal and rule-level traversal. Although the synchronization-free method generates more branch statements, the overhead is negligible compared to the performance benefits. More details are shown in Section 6.4.

Algorithm 1: Synchronization-Free Graph Traversal

```
1  Function Traversal($Graph, Operation, Condition, Result, state_{start}, state_{end}$):
2      Initialize($Graph$, $Operation$, $Condition$, $Result$, $State_{start}$, $State_{end}$)
3      $state = state_{start}$
4      while $state$ != $state_{end}$ do
5          SyncFreeTraversal($Graph$, $Operation$, $Condition$, $Result$, $state$)
6  Function SyncFreeTraversal($Graph, Operation, Condition, Result, state$):
7      if $tid$ not in 1 to $state.G.size$ then
8          return
9      $src = state.G\,[tid]$
10     for $dst$ in $src.neighbors$ do
           // Operation and condition for graph-level and rule-level traversal
11         if $src$ is Vertex and $dst$ is Vertex then $O_{v2v}(src, dst)$;
12         if $src$ is Vertex and $dst$ is Rule   then $O_{v2r}(src, dst)$;
13         if $src$ is Rule   and $dst$ is Vertex then $O_{r2v}(src, dst)$;
14         if $src$ is Rule   and $dst$ is Rule   then $O_{r2r}(src, dst)$;
15         if $dst$ is Vertex and $C_v(dst)$  then $state.G.push(dst)$;
16         if $dst$ is Rule   and $C_r(dst)$  then $state.G.push(dst)$;
```

Complexity analysis. The complexity of Algorithm 1 depends on the execution of *SyncFreeTraversal* and its execution times. The time complexity of this algorithm in the serial version is O(|V|+|R|+|E|). In the parallel version, assume the total number of threads is N. Then, the average time complexity is O((|V|+|R|+|E|)/N), which is more efficient compared to the normal non-synchronization version.

Correctness proof. The inter-level synchronization-free optimization can be directly applied to the graph traversals that can satisfy at least one of the following conditions: (1) The result field is irrelevant to the graph level. For example, each vertex uses a Boolean value to indicate whether it has been visited or not. (2) The application contains only one-level of graph traversal in each round, such as PageRank and HITS [109]. Applications that satisfy the conditions do not require the corrective procedure, so they can achieve high-performance improvement with this optimization.

We have the following proof of correctness for the inter-level synchronization-free graph traversal. (1) In the first condition, for those applications where the result field is irrelevant to the graph-level, CompressGraph needs only to ensure that the vertices that can be traversed in the original graph

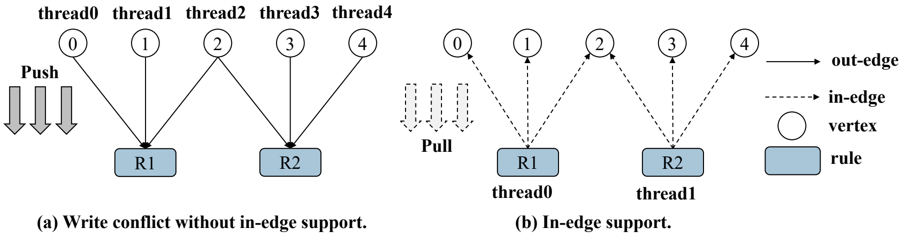


Fig. 10. An example of compressed graph direct processing based on in-edge support.

can also be traversed in the compressed graph. Given a root vertex, for any vertex v that can be traversed from the root, we denote the path from the root to v as $\{root, p_1, p_2, ..., p_n, v\}$. If all the edges $<root, p_1>$, $<p1, p2>$, ..., $<p_n, v>$ on this path exist in the compressed graph, then it is obviously possible to traverse from the root to v along the same path in the original graph. Otherwise, assume that an edge $<p_i, p_{i+1}>$ is missing in the compressed graph. Then, there must be a rule r, which is the direct or indirect parent node of p_{i+1}, and r has already been traversed when we traverse from p_i to p_{i+1}. In this situation, we can find a vertex u in the compressed graph, from which we have already visited the rule r. Accordingly, there should be a path from $root$ to u, next to r, and then to p_{i+1}. Therefore, we can traverse from the $root$ to v along the path $\{root, ..., u, r, ..., p_{i+1}, p_{i+2}, ..., p_n, v\}$. The proof is also applicable to the cases of multiple missing edges. (2) In the second condition, those applications containing only one iteration of graph-level do not involve graph-level overlap. In detail, it is unnecessary to process the vertices and rules belonging to different graph levels in one round of iterations, and thus the optimization can be directly applied.

4.4 In-Edge Support Handling Write Conflicts

To handle write conflicts in parallel, we propose a compressed graph processing solution based on in-edge design. In detail, we also store the inverted edges from the destination to the source, that is, the in-edge, which corresponds to the out edge from source to destination. With such a design, CompressGraph can put conflicting operations into one thread to avoid massive atomic operations.

General design. In graph traversal, a common operation is to pass values from vertices to neighbors. As for CompressGraph, the system can pass values, not only to vertices but also to rules. Unfortunately, multiple threads operating on the same vertex or rule at the same time can cause write conflicts. Atomic operations can ensure the correctness of this procedure, but can severely decrease performance. To solve this problem, we process the compressed graph based on the in-edge records, which store the elements pointing to the sources of vertices or rules. We use one thread to aggregate the values of all in-edges of vertices or rules, avoiding write conflicts.

Example. Figure 10 shows an example of compressed graph direct processing based on the in-edge support. Assume in the partial graph, CompressGraph needs to push values from vertices $0, 1, 2$ to $R1$, and from vertices $2, 3, 4$ to $R2$. Figure 10 (a) is an example of write conflicts without in-edge support. In detail, suppose threads 0-4 are assigned to vertices 0-4. Then, threads 0-2 write to $R1$ at the same time incurring write conflicts, so do to threads 2-4. Atomic operations can solve this problem but serialize the multi-threaded operations, degrading the benefits of parallelism. In contrast, Figure 10 (b) is an example of in-edge support, where we assign $thread0$ to $R1$ and $thread1$ to $R2$. Based on the in-edge design, $thread0$ pulls the values from neighbors 0-2 for $R1$ whereas $thread1$ pulls the values from neighbors 2-4 for $R2$. Such an in-edge design can avoid write conflicts and does not require atomic operations.

Complexity analysis. The time complexity of graph traversal based on in-edge design is O(|V|+|R|+|E|), which is the same as that of graph traversal based on out-edge design. However, for those traversal applications with write conflicts, in-edge support can save |E| atomic operations, thus improving the overall performance.

5 IMPLEMENTATION

We develop CompressGraph on both CPU and GPU platforms using C++ and CUDA. This section shows the implementation details.

Graph compression. We use the rule-based compression [142] discussed in Section 2.3 as our graph compressor. Similarly, CompressGraph recursively represents neighbors of vertices into hierarchically compressed form. There are two major differences between CompressGraph and previous works. First, the neighbors of a vertex is a set but not a sequence, which means that although two vertices have the same neighbors, their neighbors can be different in representation because of different permutations. Second, we need to ensure that we can accurately and randomly access neighbors of any vertices in the compressed state. Accordingly, we sort the sequence of neighbors of vertices to eliminate the differences in representation. In order to randomly access the neighbors of different vertices, we insert splitters between different vertices, and the number of different splitters is the number of vertices. Each splitter is unique and is inserted in the root. In this way, we can randomly obtain the neighbors of each vertex according to the position of the splitter. We then compress the processed adjacent list, whose length is the sum of the numbers of the edges and the splitters. CompressGraph recursively uses hierarchical rules to represent common neighbors and generates a set of context-free grammar (CFG) to describe the original graph. Thus, the computation complexity of compression is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph.

Note that in CompressGraph, a rule represents a set of common neighbors, and the relations of the rules can be further represented as a DAG. A possible concern could be that the rules can be ambiguous, i.e., two possible rules can be applied to the neighbors of a node. Fortunately, such an ambiguity problem can be addressed by immediately replacing the subsequence that can be expressed as a rule during the scanning of a vertex's neighbor sequence in order. For example, in Figure 4, the neighbor list of vertex 0 is {1,2,5,6,7}, where {1,2} and {2,5} can all be replaced as rules. We first replace {1,2} instead of {2,5} with a rule to obtain the neighbor sequence. Thus, CompressGraph can eliminate rule ambiguity. Therefore, after graph compression, an inner rule-based DAG is included inside, which needs further traversal. Moreover, we develop a parallel compression to significantly reduce the compression time. Note that the innovation of this paper mainly lies in how to use the rule-based representation for graph processing. Hence, we focus on the direct processing of compressed graphs by parsing rules.

Weighted graph. To support weighted graph $G <V, E, W(E)>$, we separately store the weights $\{w(v_1), w(v_2), ..., w_(v_n)\}$ of the neighbors $\{v_1, v_2, ..., v_n\}$ of vertex v. Then, we compress the neighbors $\{v_1, v_2, ..., v_n\}$ without weights. Since we sort the neighbors of the vertex before compression, the relative positions of the neighbor vertices and rules are the same as before compression. We can obtain the weight of a neighbor by calculating the position of the current neighbor in the compressed format. For a neighbor sequence $\{e_1, e_2, ..., e_m\}$ of vertex v, the weight of u_i is $W(\sum_{k=0}^{k<i} Length(e_k))$. $Length(e_k)$ is 1 if e_k is a vertex, or $Length(e_k)$ is the number of vertices included in the rule if e_k is a rule.

In detailed implementation, for the edge type of <V,V>, only one weight exists between two vertices, which is the simplest case. To represent the weight of an edge containing a rule, we propose different solutions for different types of edges, including <V,R>, <R,V>, and <R,R>. 1) For <V,R>, it contains all the weights between the vertex and all vertices within the rule. We can aggregate all the weights to obtain the weight of the edge type <V,R>, represented as $Aggregate(V, R)$. Considering that different graph applications can use weights in diverse manners, we provide several aggregate functions, including *sum*, *average*, *max*, and *min*. 2) For <R, V>, we propose a new traversal design to avoid ambiguity, since there can be multiple parent vertices of the rule. For instance, if $v1$ and $v2$ all point to $r1$, and $r1$ points to $v3$, then, the weights of <$r1$,$v3$> are ambiguous and can be interpreted as the weight of <$v1$,$v3$> or <$v2$,$v3$>. To address this problem, we record the parent vertex in a specific rule traversal and represent it as *src_root*. Thus, the weight of <$r1$,$v3$> is <*src_root*,$v3$> in a certain rule traversal. 3) For <R,R>, we combine the solutions of <V,R> and <R,V> above to represent the weights of the edge type <R,R>, which is $Aggregate(src\_root, dst)$. With these solutions, CompressGraph can support weighted graphs.

Example. In the example of single-source shortest path (SSSP) on a weighted graph, classical algorithm [33] starts from a vertex, and places all the updated vertices into the processing set until no more vertices need to be updated. For a vertex v, we update its distance to a shorter one only if there is a vertex u satisfying $u.distance+weight(u, v) < v.distance$. In CompressGraph, the operation O_{v2v} for the edge type $<V, V>$ and condition C_v for vertices operate the same as the original graph does. Otherwise, we perform rule traversal from u to v and record u as the *src_root* during the traversal. The Cr function returns true if $src\_root.distance + min\_aggregate(src\_root, r) < r.distance$ for rule r in the path from u to v. The O_{v2r} and O_{r2r} functions update the distance of rule r as the minimum of current distance of r and $src\_root.distance+max\_aggregate(src\_root, r)$. SSSP's $Or2v$ and $Ov2v$ are similar, which are used to replace the input vertex with the record *src_root*.

Attributed graph. CompressGraph can support attributed graphs by separately storing the attributes of each vertex. We can store attributes directly in the *Result* field, which can be used in the *Operation* and *Condition* functions. Since CompressGraph does not change the attributes of vertices in compression, users can easily define the attribute-based graph analytics applications. For example, for an attributed graph with a label for each vertex, the label constrained connectivity (LCC) [7, 125] algorithm detects whether there is a path between a give vertex pair $<u, v>$ where the labels of all vertices on this path belong to a given label set L. LCC adopts the BFS implementation to start from a vertex and add its neighbors whose label is in L to the processing set. Then, LCC iterates until v is found or no new vertices are added. CompressGraph can support it by adding the label constraint for C_v functions.

6 EVALUATION

6.1 Experimental Setup

Evaluated methods. We compare CompressGraph with three state-of-the-art graph compression solutions. WebGraph [4, 13] is a novel in-memory compressed graph representation. Ligra+ [117] is a lightweight graph processing framework that compresses graphs based on encoding. Gunrock [129] is a powerful GPU-based graph-centric programmable framework.

Benchmarks. We use six common graph applications, including breadth-first search (BFS), single-source shortest path (SSSP), weakly connected component (CC), PageRank (PR), topological sorting (TP), and Hyperlink-Induced Topic Search (HITS). These benchmarks have been used in various previous studies [56, 81, 99, 147, 154, 155]. We evaluate these benchmarks on CompressGraph, WebGraph, Ligra+, and Gunrock.

Datasets. We evaluate CompressGraph using 12 graphs listed in Table 1. |V| represents the number of vertices, and |E| represents the number of edges. We use CompressGraph to compress these graphs, with the "#Rules" column showing the number of rules after compression. We also report the size of the original graph and the compressed graph stored by adjacency list, denoted

Table 1. Graphs used in evaluation.

Dataset	Graph	|V|	|E|	#Rules	*Size*	$Size_c$	γ
1	uk-2007-05@100000	100,000	3,150,615	26,044	24M	5M	4.88
2	cnr-2000	325,558	3,216,151	28,748	27M	10M	2.59
3	web-BerkStan	685,231	7,600,581	92,614	63M	26M	2.43
4	in-2004	1,382,909	16,917,053	132,416	153M	44M	3.16
5	eu-2005	862,665	19,235,139	213,755	153M	43M	3.52
6	uk-2007-05@1000000	1,000,000	41,247,159	222,678	322M	47M	6.77
7	hollywood-2009	1,139,906	113,891,327	2,109,634	877M	367M	2.39
8	eu-2015-host	11,264,052	386,915,963	273,080	2.97G	871M	3.49
9	arabic-2005	22,744,080	639,999,458	864,234	4.93G	0.98G	5.03
10	it-2004	41,291,594	1,150,725,436	1,458,222	8.88G	1.74G	5.11
11	gsh-2015-host	68,660,142	1,802,747,600	2,521,370	13.94G	6.47G	2.15
12	sk-2005	50,636,154	1,949,412,601	2,372,508	14.9G	2.62G	5.68

as *Size* and $Size_c$ respectively. The compression ratio, denoted as γ, is defined as the size of the original data divided by the size of the CompressGraph-compressed data. These graphs can be downloaded from WebGraph [12, 13] and the Stanford Network Analysis Project (SNAP) [72]. They have been widely used in previous studies [65, 97, 99, 147, 154]. These graphs are originally stored in adjacency list format. WebGraph and Ligra+ need to compress the graph in advance to be used as the input of the program. For Gunrock, we convert the graph data into "mtx" format to serve as the input. Each line in the "mtx" format records the *src* and *dst* of one edge.

Platform. We evaluate CompressGraph on a GPU server. The platform is equipped with an Intel Core i9-10900X CPU, which includes ten cores with 20 threads. The global memory is 128 GB. The platform is also equipped with an Nvidia GeForce RTX 3090 GPU, powered by Ampere micro-architecture with 24 GB of G6X memory. The operating system of the platform is Ubuntu 20.04.01.

6.2 Performance Benefits

CPU performance speedup. On average, CompressGraph achieves 1.97× speedup over Ligra+ and 27.01× speedup over WebGraph. We show the detailed performance speedups compared to Ligra+ on different workloads in Figure 11, and we have the following observations. First, CompressGraph can bring significant performance improvement in most cases, which proves the effectiveness of CompressGraph on CPU. Second, CompressGraph achieves the highest performance benefits in PR and HITS. The reason is that the operation time on each edge in these two applications is long, and therefore the advantage of data reuse in CompressGraph can be fully utilized. Third, for datasets with similar compression ratios, CompressGraph can achieve greater speedup on the datasets with more edges. The reason is that the reusable part of the application with more edges accounts for a higher proportion of the total execution time. For example, the compression ratios of datasets 6 and 12 are close, but dataset 12 has more edges. Accordingly, CompressGraph achieves greater speedup on dataset 12.

GPU performance speedup. We show the performance speedup on GPU compared to Gunrock in Figure 12. For the last four large graphs, because the space required by Gunrock exceeds the GPU memory limit, we use GraphBIG [99] as the baseline. We have the following observations. First, CompressGraph outperforms Gunrock by 3.95×, which proves that CompressGraph is highly suitable for GPU parallelism. Second, due to the GPU optimization mentioned in Sections 4.3

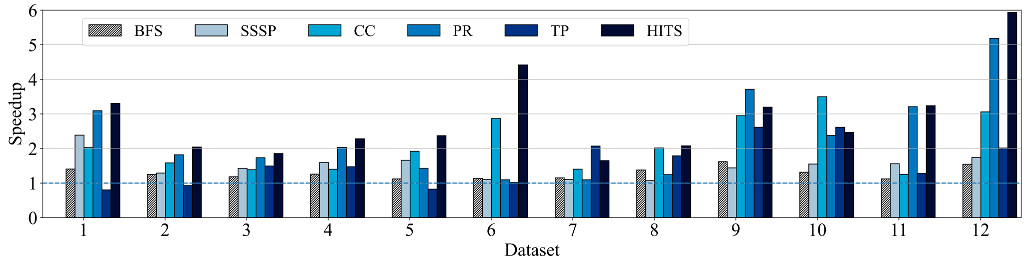


Fig. 11. CPU performance speedup (vs. Ligra+).

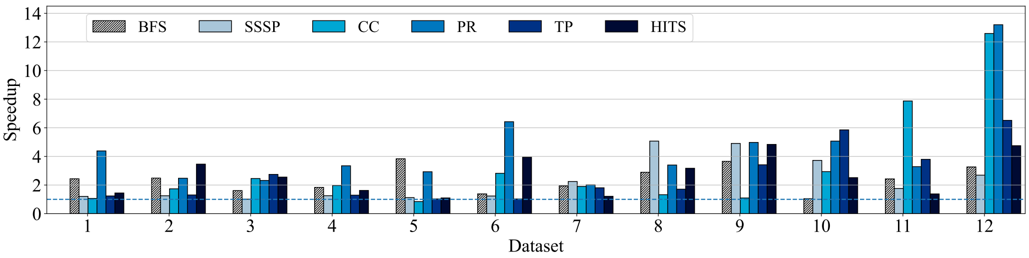


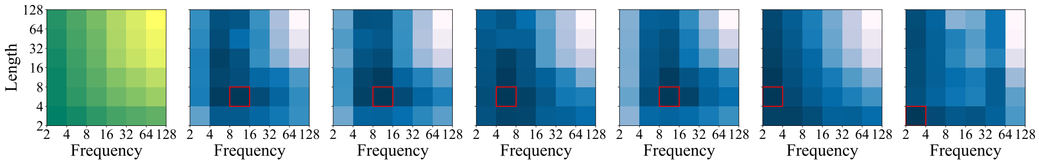
Fig. 12. GPU performance speedup (vs. Gunrock).

and 4.4, CompressGraph achieves a greater speedup on the GPU than that on the CPU. Third, CompressGraph achieves the highest speedup in PR. The reason is that PR can provide relatively high computation density, and we use the in-edge support in PR to remove the write conflicts, which enables CompressGraph to take full advantage of the data redundancy on GPU. Moreover, HITS achieves more than two times speedup, which shows that CompressGraph provides a clear performance improvement for backward traversal.

Our GPU performance also includes the transfer time between CPU and GPU. For Gunrock, the CPU-GPU transfer time accounts for 8.62% of the total runtime on average. As for CompressGraph, the CPU-GPU transfer time accounts for only 6.95% on average. When comparing the CPU-GPU transfer times of Gunrock and CompressGraph, we find that the transfer time of CompressGraph is 19.32% of Gunrock's transfer time, illustrating one of the main advantages of CompressGraph on GPU.

6.3 Space savings

Runtime space savings. CompressGraph can save 74.6% storage space savings on average. Moreover, CompressGraph can achieve 78.20% memory savings on CPU and 77.19% on GPU. CompressGraph involves marginal additional memory costs, especially compared to the original uncompressed graphs. In detail, the memory costs of CompressGraph can be divided into two parts: the compressed graph data and the *Result* field. We use a data structure similar to the adjacency list in memory to represent the compressed graph, which is consistent with its storage form on disk. Compared with other graph compression formats, CompressGraph needs additional space to store the *Result* field for rules. However, the *Result* field for rules is negligible compared to the whole memory space overhead of applications. Experiments show that the memory occupancy of CompressGraph is only 32.19% of that of the original graph. Specifically, the additional memory



(a) Compress ratio. (b) BFS. (c) SSSP. (d) CC. (e) PR. (f) TP. (g) HITS.

Fig. 14. Tradeoffs in space and time of CompressGraph. The darker the color, the better is the result.

cost from the *Result* field accounts for only 0.82% of the applications' memory space overhead based on the original graph and 2.54% based on CompressGraph. Moreover, during execution, the program needs auxiliary data structures, which also causes little space overheads. The space overhead of auxiliary data is slightly different for diverse applications on CPU/GPU platforms. For example, BFS needs to maintain a queue, but PR does not. We compare the space overhead of auxiliary data in CompressGraph with that in other graph compression formats. Experiments show that CompressGraph entails an additional space overhead of only 0.79% of the whole memory space overhead.

Time/space measurement. We next measure the time and space relations. We use the number of processed edges per second to represent performance, and the ratio of the size of the compressed graph to the number of edges, which is bits per edge (*bpe*), to represent space cost. Particularly, since CompressGraph utilizes only data redundancy, the *bpe* of a compressed graph using CompressGraph can reflect the data redundancy of the graphs. We use the average performance of CC on all datasets for illustration, as shown in Figure 13. Other workloads have similar performance behavior. We have the following observations. First, for conventional compression formats, greater space savings usually lead to increased processing overhead. However, CompressGraph can overcome this shortcoming because it can achieve high performance by utilizing redundancy to avoid repeated computation. In Figure 13, CompressGraph (CPU) generates 1.69×10^7 edges/sec under 8.81 *bpe*. Second, with similar *bpe*, utilizing GPU can deliver 40.84% extra performance benefits, which proves the efficiency of CompressGraph on GPU. Third, CompressGraph can obtain an even higher compression ratio by integrating other compression methods. In detail, CompressGraph-compressed results include vertices and rules, and the content of a rule has the same form of neighbors of the vertex. We can treat the rules as new vertices, meaning that the compressed form of CompressGraph is consistent with common graph representations. Therefore, we can apply other compression methods after CompressGraph compression. For example, after integrating WebGraph, CompressGraph achieves the smallest *bpe* in Figure 13.

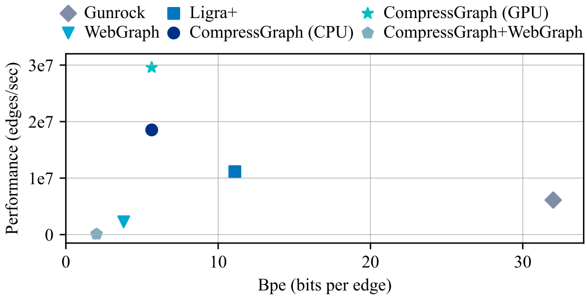


Fig. 13. Time/space measurement of different formats.

6.4 Benefit Breakdown

Dynamic rule traversal strategy. As discussed in Section 4.2, we dynamically choose to use intra-thread rule traversal or inter-thread rule traversal during the processing of graph analytics. The dynamic rule traversal strategy provides 18.74% performance improvement over the intra-thread-only rule traversal strategy, and 51.14% improvement compared to the inter-thread-only rule traversal strategy. The determination overhead accounts for only 3.8% of the total time. Moreover, to guide the selection, we need to store the number of iterations required to traverse each rule, which accounts for only 0.072% of the overall memory consumption on average.

Synchronization-free traversal. The benefits of synchronization-free traversal come from the time overlap of graph traversal and rule traversal. We evaluate the synchronization-free benefits by comparing CompressGraph with the implementation without the synchronization-free design. Experiments show that this optimization brings 42.40% benefits on average. The synchronization-free design is effective for applications that need many levels of traversal, such as BFS and SSSP. Moreover, BFS does not need the corrective procedure, which has less overhead than SSSP. Except for BFS and SSSP, HITS achieves the highest benefits. The reason is that HITS needs to traverse the in-edges and out-edges at the same time in a single round, and synchronization-free design can obtain benefits from both in-edge and out-edge traversals.

In-edge design. The in-edge design is effective in compute-intensive applications, such as PR and HITS. By removing write conflicts, this optimization leads to an average of 28.4% performance improvement. The in-edge design brings prominent performance speedup on graphs with a large number of edges per element (vertex or rule). For example, CompressGraph can achieve 53.21% and 58.06% performance improvements on datasets 7 and 8, respectively.

Branch reduction. The branch reduction design can reduce the number of operations of conditional statements. This optimization brings an average of 10.02% performance improvement on CPU and 21.31% on GPU. The performance improvement on GPU is greater than that on CPU. The reason is that the massive branches can cause severe performance degradation of GPU parallel execution.

Impacts for different rules. We compare three rule generation methods that are similar to our rule definition, TADOC [146], LZW [32], and RePair [70], whose complexities are all O(N). Their average compression ratios are 6.23, 4.78, and 4.92. We also compare the performance on top of the compressed data of these three compression methods. The result shows that CompressGraph with TADOC is 1.3× faster than that with LZW, and 1.26× faster than that with RePair. Therefore, we build our compressor on TADOC.

6.5 Design Tradeoffs in Space and Time

In this part, we show our design tradeoffs in space and time. The generation of optimal rules is an NP-hard problem, as described in the previous study [22]. Hence, the compressor does not pursue the optimal rule compression, but adopts the simplification [100] to ensure that the compression result can be obtained in linear time. However, we find that rule length and rule frequency are still adjustable to ensure a balance in time and space. Rule length refers to the minimum number of elements to store per rule. Rule frequency refers to the minimum number of times that a rule appears in the compressed graph. For a given graph, we can set thresholds of rule length and frequency adapting to different scenarios. Lower thresholds usually mean more rules to be generated. However, there is a tradeoff between time savings and space savings. We find that it is not true that the smaller the compressed graph, the greater the benefit of data reuse for computation savings. We show our findings in Figure 14. Figure 14 (a) shows the space benefits of different length and frequency thresholds. The highest performance appears in the lower-left-hand corner of

Figure 14 (a). However, in Figures 14 (b) to (g), the rule length and frequency setting for the highest performance do not appear in the lower-left-hand corner of the performance heat map. Hence, we set the rule length and frequency according to the tradeoffs from both time and space perspectives.

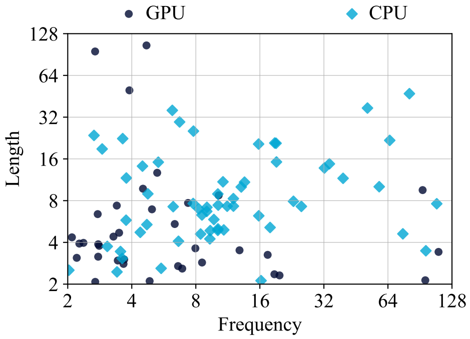


Fig. 15. Configuration distribution at the highest performance.

Configuration. We show the rule length and frequency for the highest performance with different workloads on both CPU and GPU in Figure 15. To select the rule length and frequency, we first compress the graph G_{init} with parameters (2,2). Then, we resort to a multi-layer perception (MLP) [105] model to assist in determining the rule length and frequency configuration. In detail, we abstract the rule configuration as a classification problem, where each category corresponds to a combination of parameters of rule length and rule frequency. The input features include the number of vertices, the number of edges, the length and frequency distributions of rules in G_{init}, the available threads, and a feature vector indicating different workloads from processing the sampled data. Note that the workload features can be obtained from processing a small sampled graph. As for model setting, we construct a three-layer neural network with 1500 hidden neurons in each hidden layer, using $Tanh$ as the activation function; we use $softmax$ for output, and cross-entropy as loss function. The model attains 82.5% accuracy. Besides, the difference between our selection and the highest result for different configurations is only 12.42%, which is acceptable. In terms of efficiency, we can complete the inference in less than 7.18 ms, which is very efficient.

6.6 Additional Baselines and Discussion

There are many other novel graph systems, such as disk-based graph systems [38, 67, 87, 97] and graph systems built on relational databases [35, 61, 121, 136, 137], targeting different application scenarios. In this section, we compare CompressGraph with the representatives of these systems and show our application scenario.

Comparison with the disk-based graph system. Disk-based graph systems [38, 67, 87, 97] store the entire graph on disk and a portion of it in memory to handle large graphs. HCon [38] is a novel hierarchical contraction scheme for querying large graphs, which recursively contracts a set of vertices satisfying a particular structure into supernodes until the graph size fits in the memory limitation. We compare CompressGraph with HCon in terms of space cost and performance on PR and CC. Since HCon is not open source, we implement a version using clique, star, and diamond (the most frequent regular structures in our dataset). HCon can save an average of 72.2% graph size when the contract level is 3, while CompressGraph can save 74.6% graph size. HCon can increase the number of levels to further reduce memory space consumption. However, increasing the number of levels causes performance degradation, and its top-level graph is already small enough to fit in memory. The average performance of HCon on PR and CC is 3.73× and 1.06× slower than that of CompressGraph. HCon needs to decontract from disk on PR, which is significantly slower than accessing data from memory. HCon can handle CC well. The reason is that all the vertices in the structures used in HCon are connected, so there is no need to decontract in CC. Note that HCon uses a different design philosophy compared to CompressGraph. The novelty of HCon is the optimization of disk access for graph queries. In contrast, we assume that the compressed graph can fit into memory. Therefore, we can apply HCon to CompressGraph to handle compressed graphs that cannot fit in memory.

Comparison with graph systems built on relation databases. In the real world, large amounts of data are stored in relational databases. Many works [35, 61, 121, 136, 137] focus on extracting and analyzing graphs from relational databases. GraphGen [136, 137] is a novel system enabling efficient graph analytics on top of relational data. In particular, GraphGen proposes a condensed representation for the graph data extracted from the relational database to eliminate the duplication issue. After we compare GraphGen with CompressGraph, we find that GraphGen cannot gain the same space savings as CompressGraph. The reason is that there is no duplication defined by GraphGen in standard graph we use. We also evaluate the performance of GraphGen on BFS and PR. Experiments show that GraphGen is 1.89× slower than CompressGraph on BFS, and 2.84× slower than CompressGraph on PR. Note that the application scenario of GraphGen is graph analytics of relational data, which is orthogonal to CompressGraph.

Weighted graph. To measure the performance of CompressGraph on weighted graphs, we generate random weights for the graphs in Table 1. We use SSSP as an example and measure the performance benefits of compressed weighted graphs in SSSP. Experiments show that we can still obtain 27.52% space savings and 1.93× performance speedup on average.

Attributed graph. We evaluate the performance of CompressGraph on attributed graphs. We generate random labels for each vertex in graphs (four types in total). The experiments on LCC show that we can obtain 72.58% storage space savings and 1.55× performance speedup on average.

Application scope. Similar to WebGraph and Ligra+, CompressGraph is designed for graph analytics on datasets that do not change frequently. The compressed graph can be accessed and used by many users for various purposes. For example, some users may need to perform a *page rank*, whereas others compute the *single-source shortest path.* With CompressGraph, a graph needs to be compressed only once, and thus, the compression time is amortized by the time savings in the many uses of the graphs by many users. Different from HCon [38], to support maintenance, CompressGraph uses a bitmap to indicate the updated vertices and edges, and saves the changes to another data structure. Recompression occurs when the system is idle or the modification is over 25% of the compressed graph. We also measure the compression time required by different methods. On average, the compression time of CompressGraph is 31.29% shorter than that of Ligra+, and 45.27% shorter than that of WebGraph, respectively.

7 RELATED WORK

Compression, as an important data storage optimization, has been applied in various fields [15, 25, 46, 47, 68, 69, 73–78, 84, 85, 89, 112, 138]. This section presents the related work of graph compression and processing.

Graph compression formats and systems. Many works focus on graph compression [4, 12, 13, 16, 26, 29, 34, 57, 83, 86, 115, 116, 120]. Traditionally, graphs are stored in adjacency matrices and adjacency lists. The adjacency matrix has high access efficiency, but can incur a large space overhead for sparse graphs. Much effort has been invested in solving the sparsity problem of adjacency matrix [16, 34, 86, 120]. Another optimization targets data redundancy in graph storage, especially when vertices share many neighbors. It is usually performed on adjacency lists, saving the neighbors for each vertex. WebGraph [12, 13] is a novel representation of compressed graphs that provide efficient graph processing. Chierichetti *et al.* [26] applied WebGraph to social networks and further proposed methods to obtain a better ranking for neighbors [4, 57, 115]. Ligra+ [116, 117] is a novel framework that can process compressed graphs efficiently. Maneeth *et al.* [88] summarized the grammar-based compression of three types of objects: strings, trees, and hyper graphs. Maneth *et al.* [90, 91] proposed GraphRePair, which is a novel grammar-based compression method for hypergarphs. GraphRePair can detect recurring hyperedges in the hypergraph and recursively replace them with grammar rules. Zhou *et al.* [153] proposed adaptive partitioning to process graphs efficiently in large geo-distributed scales. Different from these works, CompressGraph emphasizes the reuse of intermediate results during traversal and is suitable for highly parallel application scenarios, such as GPUs.

Direct processing on compressed data. Compressed data direct processing has been proved to be a promising technology solving big data challenges [76–78, 80]. Many works utilize grammar compression on strings [14, 18, 27, 30, 51, 52, 92, 135, 147]. TADOC [107, 139, 142–146, 151] is a representative grammar-based compression method for text analytics. Traditional methods for queries on compressed data are based on suffix arrays, trees, and indexes [11, 28, 40–43, 53, 58, 102]. Succinct [1] is a representative method for queries on compressed data. Moreover, there is a large body of literature on compressing the inverted index [98, 100, 111, 114, 118, 124, 126]. CompressDB [141] enables the rule-based compression in storage layer, supporting diverse databases. Different from these works, CompressGraph targets analytics of rule-based compressed graphs, and provides efficient CPU and GPU implementations. In future work, we plan to offload part of ComprssGraph to the novel GPU-centric SmartNIC FpgaNIC [131] to enable efficient distributed graph analytics between distributed GPUs.

Graph compression for queries. Numerous studies [2, 37, 60, 94, 95] focus on graph compression for queries. Maserrat *et al.* [94] proposed a multi-position linearization compression approach to accommodate out- and in-neighbor queries. Fan et al. [37] presented a novel query-preserving graph compression for readability and pattern-matching queries. Khandelwal *et al.* [62] developed ZipG, which is an efficient distributed memory graph store based on Succinct [1], supporting interactive queries on compression format. Fan *et al.* [38] proposed a novel hierarchical contraction scheme for querying large graphs, which recursively contracts graph vertices into supernodes until the graph size can fit in the memory constraints. Xirogiannopoulos *et al.* [136] proposed a condensed graph representation to extract and analyze the graphs stored in multiple tables in a relational database. HBMax [24] proposes a compression approach to reduce the memory footprint for parallel influence maximization on multi-core architectures, by compressing the intermediate reverse reachability information and directly querying on the compressed data. In contrast, CompressGraph focuses on supporting diverse graph analytics that require scanning the whole graph through rule parsing on both CPU and GPU.

8 CONCLUSION

This paper has proposed a method to directly process compressed graphs, and developed a graph analytics engine, called CompressGraph. By enabling direct processing on compressed graphs, we can obtain 74.6% space savings, 1.97× performance improvement on CPU, and 3.95× on GPU. This paper unveils how to compress graphs by rules to make the compressed data suitable for direct processing. Then, the paper proposes data reuse methods for different types of graph applications. Moreover, the paper proposes a series of GPU optimizations to handle the parallelism challenge of directly processing without decompression on GPUs. Experiments demonstrate the huge application prospects of direct processing on compressed graph data.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (62172419, 61732014, 62072458, and U20A20226), CCF-Tencent Open Research Fund, Beijing Nova Program, the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China (21XNH182). Feng Zhang is the corresponding author of this paper.

REFERENCES

[1] 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.

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

[3] Ahmed Al-Baghdadi and Xiang Lian. 2020. Topic-based Community Search over Spatial-Social Networks. *Proc. VLDB Endow.* 13, 11 (2020), 2104–2117. http://www.vldb.org/pvldb/vol13/p2104-al-baghdadi.pdf

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

[5] 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.

[6] Prithu Banerjee, Wei Chen, and Laks VS Lakshmanan. 2019. Maximizing welfare in social networks under a utility driven influence diffusion model. In *Proceedings of the 2019 International Conference on Management of Data*. 1078–1095.

[7] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav Marathe, and Dorothea Wagner. 2008. Engineering label-constrained shortest-path algorithms. In *International conference on algorithmic applications in management*. Springer, 27–37.

[8] 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.

[9] Maciej Besta and Torsten Hoefler. 2018. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv preprint arXiv:1806.01799* (2018).

[10] Song Bian, Qintian Guo, Sibo Wang, and Jeffrey Xu Yu. 2020. Efficient Algorithms for Budgeted Influence Maximization on Massive Social Networks. *Proc. VLDB Endow.* 13, 9 (2020), 1498–1510. https://doi.org/10.14778/3397230.3397244

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

[12] 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.

[13] 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.

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

[15] Mireille Bousquet-Mélou, Markus Lohrey, Sebastian Maneth, and Eric Noeth. 2015. XML compression via directed acyclic graphs. *Theory of Computing Systems* 57, 4 (2015), 1322–1371.

[16] 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.

[17] 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.

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

[19] Chengliang Chai, Guoliang Li, Jian Li, Dong Deng, and Jianhua Feng. 2016. Cost-Effective Crowdsourced Entity Resolution: A Partial-Order Approach. In *SIGMOD*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 969–984.

[20] Chengliang Chai, Guoliang Li, Jian Li, Dong Deng, and Jianhua Feng. 2018. A partial-order-based framework for cost-effective crowdsourced entity resolution. *VLDB J.* 27, 6 (2018), 745–770.

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

[22] 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.

[23] Hongzhi Chen, Xiaoxi Wang, Chenghuan Huang, Juncheng Fang, Yifan Hou, Changji Li, and James Cheng. 2019. Large Scale Graph Mining with G-Miner. In *Proceedings of the 2019 International Conference on Management of Data*. 1881–1884.

[24] Xinyu Chen, Marco Minutoli, Jiannan Tian, Mahantesh Halappanavar, Ananth Kalyanaraman, and Dingwen Tao. 2022. HBMax: Optimizing Memory Efficiency for Parallel Influence Maximization on Multicore Architectures. *arXiv preprint arXiv:2208.00613* (2022).

[25] 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.

[26] 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.

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

[28] 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.

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

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

[31] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. 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, 377–392. https://doi.org/10.1145/3318464.3386144

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

[33] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[34] 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.

[35] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. 2015. The Case Against Specialized Graph Analytics Engines.. In *CIDR*.

[36] 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

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

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

[43] 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.

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

[45] Jun Gao, Jiazun Chen, Zhao Li, and Ji Zhang. 2021. ICS-GNN: Lightweight Interactive Community Search via Graph Neural Network. *Proc. VLDB Endow.* 14, 6 (2021), 1006–1018. http://www.vldb.org/pvldb/vol14/p1006-gao.pdf

[46] 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.

[47] Adrià Gascón, Markus Lohrey, Sebastian Maneth, Carl Philipp Reh, and Kurt Sieber. 2020. Grammar-based compression of unranked trees. *Theory of Computing Systems* 64, 1 (2020), 141–176.

[48] Advitya Gemawat. 2021. GraphGem: Optimized Scalable System for Graph Convolutional Networks. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2920–2922. https://doi.org/10.1145/3448016.3450573

[49] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin C. Herbordt. 2020. AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 922–936. https://doi.org/10.1109/MICRO50266.2020.00079

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

[51] 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.

[52] 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.

[53] 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.

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

[55] 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.

[56] 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.

[57] 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*.

[58] 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.

[59] 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.

[60] 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.

[61] Alekh Jindal, Samuel Madden, Malú Castellanos, and Meichun Hsu. 2015. Graph analytics using vertica relational database. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 1191–1200.

[62] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. 2017. Zipg: A memory-efficient graph store for interactive queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1149–1164.

[63] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 239–252.

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

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

[66] Seongyun Ko, Taesung Lee, Kijae Hong, Wonseok Lee, In Seo, Jiwon Seo, and Wook-Shin Han. 2021. iTurboGraph: Scaling and Automating Incremental Graph Analytics. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 977–990. https://doi.org/10.1145/3448016.3457243

[67] 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.

[68] 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.

[69] 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.

[70] N Jesper Larsson and Alistair Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732.

[71] David Lee and Mihalis Yannakakis. 1996. Principles and methods of testing finite state machines-a survey. *Proc. IEEE* 84, 8 (1996), 1090–1123.

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

[73] 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.

[74] 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.

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

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

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

[78] 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.

[79] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling distance labeling on small-world networks. In *Proceedings of the 2019 International Conference on Management of Data*. 1060–1077.

[80] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 1–13.

[81] 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.

[82] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based Community Search over Large Directed Graphs. 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, 2183–2197. https://doi.org/10.1145/3318464.3380587

[83] 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.

[84] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. 2013. XML tree structure compression using RePair. *Information Systems* 38, 8 (2013), 1150–1167.

[85] Markus Lohrey, Sebastian Maneth, and Carl Philipp Reh. 2017. Compression of unordered XML trees. In *20th International Conference on Database Theory (ICDT 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

[87] 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.

[88] Sebastian Maneth. 2019. Grammar-Based Compression. *Journal: Encyclopedia of Big Data Technologies* (2019), 801–808.

[89] Sebastian Maneth and Fabian Peternek. 2015. A survey on methods and systems for graph compression. *arXiv preprint arXiv:1504.00616* (2015).

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

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

[92] Sebastian Maneth and Fabian Peternek. 2020. Constant delay traversal of grammar-compressed graphs with bounded rank. *Information and Computation* 273 (2020), 104520.

[93] Davide Marengo, Cornelia Sindermann, Jon D Elhai, and Christian Montag. 2020. One social media company to rule them all: Associations between use of Facebook-owned social media platforms, sociodemographic characteristics, and the Big Five personality traits. *Frontiers in psychology* 11 (2020), 936.

[94] 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.

[95] 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.

[96] 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.

[97] 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})*.

[98] Kinichi Mitsui. 1993. Information retrieval based on rank-ordered cumulative query scores calculated from weights of all keywords in an inverted index file for minimizing access to a main database. US Patent 5,263,159.

[99] 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.

[100] 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.

[101] 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.

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

[103] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Progressive Top-K Nearest Neighbors Search in Large Road Networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20).* Association for Computing Machinery, New York, NY, USA, 1781–1795. https://doi.org/10.1145/3318464.3389746

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

[105] Sankar K Pal and Sushmita Mitra. 1992. Multilayer perceptron, fuzzy sets, classifiaction. (1992).

[106] 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.

[107] Zaifeng Pan, Feng Zhang, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. Exploring data analytics without decompression on embedded GPU systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 7 (2021), 1553–1568.

[108] Alberto Parravicini, Rhicheek Patra, Davide B. Bartolini, and Marco D. Santambrogio. 2019. Fast and Accurate Entity Linking via Graph Embedding. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Amsterdam, The Netherlands, 30 June 2019*, Akhil Arora, Arnab Bhattacharya, and George H. L. Fletcher (Eds.). ACM, 10:1–10:9. https://doi.org/10.1145/3327964.3328499

[109] Punit Patel and Kanu Patel. 2015. A Review of PageRank and HITS Algorithms. *International Journal of Advance Research in Engineering, Science & Technology (IJAREST)* 2, 1 (2015), 1–4.

[110] Jiezhong Qiu, Laxman Dhulipala, Jie Tang, Richard Peng, and Chi Wang. 2021. LightNE: A Lightweight Graph Processing System for Network Embedding. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2281–2289. https://doi.org/10.1145/3448016.3457329

[111] Erhard Rahm and Hong Hai Do. 2000. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.* 23, 4 (2000), 3–13.

[112] 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.

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

[114] Kunihiko Sadakane. 2007. Compressed suffix trees with full functionality. *Theory of Computing Systems* 41, 4 (2007), 589–607.

[115] 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.

[116] 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.

[117] 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.

[118] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. 2014. *Operating system concepts essentials.* Wiley Hoboken.

[119] 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.

[120] 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.

[121] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* 1887–1901.

[122] Yizhou Sun. 2020. Graph Neural Networks for Graph Search. In *GRADES-NDA'20: Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Portland, OR, USA, June 14, 2020*, Akhil Arora, Semih Salihoglu, and Nikolay Yakovets (Eds.). ACM, 1:1. https://doi.org/10.1145/3398682.3399159

[123] Frank Tetzel, Romans Kasperovics, and Wolfgang Lehner. 2019. Graph Traversals for Regular Path Queries. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Amsterdam, The Netherlands, 30 June 2019*, Akhil Arora, Arnab Bhattacharya, and George H. L. Fletcher (Eds.). ACM, 5:1–5:8. https://doi.org/10.1145/3327964.3328494

[124] Stephen Lyle Tu, M Frans Kaashoek, Samuel R Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. (2013).

[125] Lucien DJ Valstar, George HL Fletcher, and Yuichi Yoshida. 2017. Landmark indexing for evaluation of label-constrained reachability queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 345–358.

[126] Flavian Vasile, Elena Smirnova, and Alexis Conneau. 2016. Meta-prod2vec: Product embeddings using side-information for recommendation. In *Proceedings of the 10th ACM Conference on Recommender Systems*. 225–232.

[127] Alina Vretinaris, Chuan Lei, Vasilis Efthymiou, Xiao Qin, and Fatma Özcan. 2021. Medical Entity Disambiguation Using Graph Neural Networks. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 2310–2318. https://doi.org/10.1145/3448016.3457328

[128] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, and Zhenyu Guo. 2021. APAN: Asynchronous Propagation Attention Network for Real-time Temporal Graph Embedding. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2628–2638. https://doi.org/10.1145/3448016.3457564

[129] 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.

[130] Ye Wang, Qing Wang, Henning Koehler, and Yu Lin. 2021. Query-by-Sketch: Scaling Shortest Path Graph Queries on Very Large Networks. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1946–1958. https://doi.org/10.1145/3448016.3452826

[131] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. 2022. FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 967–986.

[132] Stanley Wasserman, Katherine Faust, et al. 1994. Social network analysis: Methods and applications. (1994).

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

[134] Jack Waudby, Benjamin A. Steer, Arnau Prat-Pérez, and Gábor Szárnyas. 2020. Supporting Dynamic Graphs and Temporal Entity Deletions in the LDBC Social Network Benchmark's Data Generator. In *GRADES-NDA'20: Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Portland, OR, USA, June 14, 2020*, Akhil Arora, Semih Salihoglu, and Nikolay Yakovets (Eds.). ACM, 8:1–8:8. https://doi.org/10.1145/3398682.3399165

[135] 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.

[136] Konstantinos Xirogiannopoulos and Amol Deshpande. 2017. Extracting and analyzing hidden graphs from relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 897–912.

[137] Konstantinos Xirogiannopoulos, Udayan Khurana, and Amol Deshpande. 2015. Graphgen: Exploring interesting graphs in relational data. *Proceedings of the VLDB Endowment* 8, 12 (2015), 2032–2035.

[138] 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.

[139] Feng Zhang, Yihua Hu, Haipeng Ding, Zhiming Yao, Zhewei Wei, Xiao Zhang, and Xiaoyong Du. 2022. Optimizing random access to hierarchically-compressed data on GPU. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 233–247.

[140] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. G-TADOC: Enabling efficient GPU-based text analytics without decompression. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1679–1690.

[141] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. 2022. CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases. In *Proceedings of the 2022 International Conference on Management of Data*. 1655–1669.

[142] 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.

[143] 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.

[144] 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.

[145] 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.

[146] 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.

[147] 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).

[148] Wentao Zhang, Xupeng Miao, Yingxia Shao, Jiawei Jiang, Lei Chen, Olivier Ruas, and Bin Cui. 2020. Reliable Data Distillation on Graph Convolutional Network. 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, 1399–1414. https://doi.org/10.1145/3318464.3389706

[149] Wentao Zhang, Yu Shen, Yang Li, Lei Chen, Zhi Yang, and Bin Cui. 2021. ALG: Fast and Accurate Active Learning Framework for Graph Convolutional Networks. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2366–2374. https://doi.org/10.1145/3448016.3457325

[150] Xiaofei Zhang, M. Tamer Özsu, and Lei Chen. 2020. ELite: Cost-effective Approximation of Exploration-based Graph Analysis. In *GRADES-NDA'20: Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Portland, OR, USA, June 14, 2020*, Akhil Arora, Semih Salihoglu, and Nikolay Yakovets (Eds.). ACM, 6:1–6:10. https://doi.org/10.1145/3398682.3399164

[151] Yu Zhang, Feng Zhang, Hourun Li, Shuhao Zhang, and Xiaoyong Du. 2023. CompressStreamDB: Fine-Grained Adaptive Stream Processing without Decompression. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE.

[152] Jianlong Zhong and Bingsheng He. 2013. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2013), 1543–1552.

[153] 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)*. 2818–2830. https://doi.org/10.1109/ICDE53745.2022.00256

[154] 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.

[155] 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.

[156] Yuqing Zhu, Jing Tang, and Xueyan Tang. 2020. Pricing Influential Nodes in Online Social Networks. *Proc. VLDB Endow.* 13, 10 (2020), 1614–1627. https://doi.org/10.14778/3401960.3401961

Received April 2022; revised July 2022; accepted August 2022