Vol.2022-DBS-175 No.15
Vol.2022-IFAT-148 No.15
2022/9/9

Towards Constructing Destination Node Index for Repetition Paths

Kazuma Kusu[1,a)] Takahiro Komamizu[2,b)] Kenji Hatano[3,c)]

Abstract: Graph database management systems (GDBMSs) enable users to traverse one edge at a fixed computing cost for vast and complex graph data. However, GDBMSs cannot avoid reaching already-scanned nodes from different starting nodes by repeatedly traversing edges, which we name a repetition path, with a specific relationship type. Therefore, when a GDBMS reaches a high degree node (HDN), the number of graph traversals increases in proportion to the number of its adjacent nodes. Consequently, the cost of traversing repetition paths extremely increases affected by HDNs in conventional GDBMSs. In this paper, we propose a graph index structure to repeatedly traverse edges belonging to a specific relationship type by distinguishing between HDNs and other nodes. Moreover, we also propose a method for compressing our index to take advantage of the characteristics of real-world networks so that a graph index tends to be the easily huge size of index files.

1. Introduction

Graph is a flexible data structure which represents relationships between pieces of data, and it appears in various fields, such as social networks, science collaboration, hyperlinks on WWW, etc [1]. There are multiple demands for searching and analyzing such connected data based on the relationships in data-centric science. A graph, a data structure, can directly represent relational data consisting of nodes (a.k.a. points, vertexes) and edges (a.k.a. lines, links). A node can represent an entity (e.g. person, paper, etc.), and an edge can represent a relationship between entities (e.g. friendship, citation, etc.) In addition to the two primary elements described above, elements expressing extended information like a property and a label enable us to represent various forms of graph. Such flexible graphs like labeled graph, weighted graph, multi-graph, can adapt successfully to data dealing on various applications [2].

Researchers and companies have studied graph database management systems (GDBMSs) [3, 4] to manage a huge graph expertly and process graph traversals efficiently. Moreover, each GDBMS has a data structure for rapidly obtaining edges or nodes called a graph index or an adjacent list. A GDBMS-based *relation* structure, which is also known as table structure, is good performance at global search of sub-graphs with indexes, but the size of index tends to become huge. Such GDBMS is called non-native one because the data store do not shape up graph-like. Contrary, a native GDBMS employs a storage that manages nodes with pointers referencing their adjacent-nodes; this data store keeps a shape of a graph on storage. Therefore, native GDBMSs have an impactive characteristic for efficient traversal, called *index-free adjacency*, in a local search because the cost for searching sub-graphs depends only on an adjacent list of each node. In this paper, we presupposes employing a native GDBMS because our study considers that it requires to ensure the efficiency of graph traversal.

However, GDBMSs execute a graph traversal indiscriminately without considering the degrees of nodes which are the numbers of nodes connected from them. Hence, this traversal approach can be inefficient, because the number of graph traversal increases enormously due to the presence of high-degree nodes. To realize efficient graph traversal, graph traversal beginning from a node connected an enormous number of edges should be avoided.

In the research field of network science, real-world networks have graph-topological properties named *scale-free* or *small-world networks*. Scale-free property is a characteristic of a graph that the small number of nodes have significantly higher degree than other nodes. Small-world property (a.k.a. six degrees of separation in social networks) is another property that the average shortest path lengths between every pairs of nodes is small because nodes having huge degrees are easily reachable from other nodes. Therefore, when traversing a graph in a repeated manner, high-degree nodes are easily reachable.

[1] Doshisha University, Graduate School of Culture and Information Science, Kyotanabe, Kyoto, Japan
[2] Nagoya University, Mathematical and Data Science Center, Nagoya, Aichi, Japan
[3] Doshisha University, Faculty of Culture and Information Science, Kyotanabe, Kyoto, Japan
[a)] kkusu@acm.org
[b)] taka-coma@acm.org
[c)] khatano@mail.doshisha.ac.jp

On the contrary, GDBMSs often call an operation performing repeated traversal when users want to obtain paths consisting of edges labeled a few types of specific relationship like "a friend-of-a-friend" relationship. In the above case, a chance to reach high-degree nodes is raised due to the small-world property. Therefore, it is easily surmised that the number of candidate nodes for next traversals significantly is increased so that the performance of repeated traversals can be largely degraded.

In this study, our goal is to solve the bottleneck of a GDBMS, which causes when it repeatedly traverses edges in the graph due to the existing networks' properties. Our previous study [5] defined a repetition path as long paths obtained by repeatedly traversing edges labeled a specified relationship, and moreover, the process querying the paths was called repeated traversal. Furthermore, our previous study proposed a structure of a graph index for scanning paths specified length from a high-degree node to avoid the bottleneck process [5]. However, the proposed index had two shortcomings when the graph size becomes large. One is that the performance of scanning repetition paths is unstable, and the other is that the construction time of the index becomes unacceptably large. Therefore, in this paper, we propose a method for compressing the size of our graph index for scanning destination nodes of repetition paths, then it simultaneously enables us to construct our index faster.

Our contributions in this study is to:

- make more efficient to construct our index [5] by a compression approach; and
- enable GDBMSs to traverse long paths by a low cost.

As we described above, our index enables GDBs to improve the performance for a repeat graph traversal operation, which tends to be a high cost in real-world graphs.

2. Related Works

A considerable number of studies have been conducted on graph management methods and indexes for improving traversing performances [6]. This section discusses studies focusing on native GDBMSs employing *index-free adjacency* and those concerning indexes for searching paths in a query.

Section 2.1 describes management approach for a huge graph on native GDBMSs that employ index-free adjacency. Section 2.2 introduces indexing approaches for inquiring paths fast. Finally, in Section 2.3, we advocate the differences of our approach compared with conventional approaches. in the light of conventional ones.

2.1 The Index-free adjacency of native GDBMSs

Native GDBMSs with index-free adjacency [7] can efficiently retrieve adjacent nodes by using a small adjacent list of each node. This adjacent list of each node has pointers referring to the address of its adjacent nodes in database storage. Moreover, there is the case that the nodes' edges consist of a large variety of labeled edges, but native GDBMSs do not affect the performance of a graph traversal in such a case. This is because native GDBMSs have a hash map of edges grouped by their relationship so these can get edges having the relationship specified in a query from the hash map. To retrieve matching sub-graphs, furthermore, a native GDBMS first scans origin nodes as sub-graphs from a database; subsequently, it consecutively expands them by traversing edges based on adjacent lists of the target nodes.

In contrast, conventional native GDBMSs are not efficient for queries requiring larger hops of traversals. They only grasp adjacent connectivity of nodes, in other words, they cannot grasp connectivity of nodes which are more than one-hop away from each other. When the conventional GDBMSs process the queries requiring larger hops of traversals, they are affected by high-degree nodes because such nodes explosively increase the number of paths being traversed.

However, conventional native GDBMSs cannot grasp the all connectivities between nodes in a graph, which cannot be revealed using anything but the adjacency list of each node.

Consequently, it occasionally happens to traverse the edges even if the target node for traversing is a high-degree node. Hence, it traverses edges if the target node is that node nevertheless. As mentioned above, index-free adjacency is the most powerful characteristic for traversing the graph regionally. However, in the case of traversing the graph globally, this is not an efficient approach.

2.2 Path index

Path index-based approaches are roughly classified into two categories of approaches. Those in the first category are that DB administrators construct indices by their estimation of graph pattern queries in the future. Those in the other category assume to use query logs to extract frequent graph patterns for indexing [8].

There are two ways for managing the graph: one is to handle a huge graph, the other is to perform numerous small graphs [8]. Example graphs of the former methods are social graphs, the Web graph, coauthor graphs, citation graphs, etc. By contrast, the latter is applicable in the case of chemical structures and protein-protein interaction networks.

Most conventional graph indexes have been focused on GDBMSs for numerous small graphs. However, in the case of managing a large single graph, graph index with same way as numerous small graphs is inefficient due to its size. In a large single graph, the number of nodes and edges is larger, therefore, the number of paths in the graph is also larger than that in the numerous small graphs. As a result, the index size of the graph becomes considerably large due to the large number of path candidates, and, hence, the index scan becomes inefficient.

2.3 Our goal

Conventional approaches for extracting sub-graphs have been realized by graph traversals and the graph index described in Section 2.1 and 2.2. In the case of management for the graph, it is efficient to extract sub-graphs with index-free adjacency regionally, but native GDBMSs have not considered an inefficiency affected by nodes having too many node

degrees. Hence, in this study, we should deal with nodes having a large number of edges labeled one type of relationship. In the case of using a graph index, on the other hand, indexing a specified pattern of a path and extracting sub-graphs with the index becomes inefficient caused by the graph size. Thus, these approaches suffer from extracting sub-graphs from a large single graph in an efficient way.

In the research field of network science, researchers have reported that real-world networks have properties named scale-free and small world [9]. The scale-free property implies that the degree distribution of such a network follows a power-law distribution. The small world property indicates that arbitrary two nodes in a networks can be reached within a small number of hops. Conventional native GDBMSs traverse edges with regardless of node's degree because these do not take into account the aforementioned properties of real-world networks. Moreover, it gradually becomes easy to reach high-degree nodes by repeatedly traversing edges because of the small world property.

In this study, therefore, we aim to enable native GDBMSs to efficiently traverse a long path extracted by traversing edges labeled one type of relationship repeatedly from a huge real-world graph. We pay attention to the fact that traversing repetition paths makes the opportunity to reach the nodes from other nodes increase because of the properties of real-world graphs such as scale-free and small-world. Therefore, we propose an approach for managing the graph that enables native GDBMSs to make it efficient to traverse repetition paths by considering the nodes.

3. Proposed Method

Our study aims to improve the efficiency when GDBMSs repeatedly traverse long paths extracted by the traversing edges labeled by one type of relationship from a prominent actual graph such as social networks that grows larger with time. In this paper, we name a high-degree node *HDN* which overlaps with another concept called *hub* in network science [1]. In contrast, we define the exclusive nodes of HDNs, low-degree nodes called LDNs. We focus on the fact that traversing *repetition paths* from other nodes increases the probability of reaching to HDNs because of the properties such as scale-free and small-world of the real-world graphs. Therefore, native GDBMSs aiming to be versatile at managing a variety of graphs should be able to grasp whether a node is an HDN or not because lacking this ability would render them inefficient at dealing with HDNs.

Our previous study proposed a graph index (PR-index) for repeatedly traversing edges by collecting repetition paths, consisting of edges labeled the same relationship, from HDNs. RP-index enabled a GDBMS to traverse repetition paths faster than one without our index. However, RP-index included redundancy in its structure that repetition paths beginning from a specified HDN are also contained other HDNs. Moreover, expected results of queries with repetition paths generally are destination nodes of these paths.

In this paper, to rid redundancy of RP-index, we propose

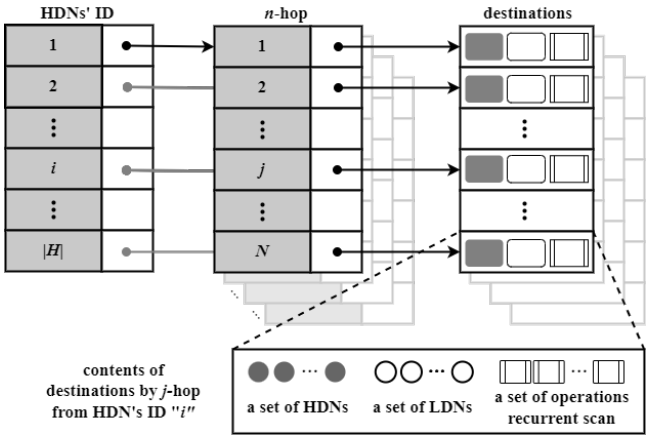

Fig. 1: This figure is a schema of RPD-index.

an algorithm for indexing compressed RP-index by judging whether reached node is HDN or LDN. Subsequently, our approach memories an operation of scanning RP-index for the ID of HDN and the length of hops into RP-index to enable GDBMSs to obtain destinations required a query later.

3.1 A schema of compressed RPD-index

RP-index can become fast to query repetition paths, such as "`-[Relation*2..4]->`"[*1] in a graph query language like Cypher, PGQL, G-core, and GQL, but indexing these paths generate a huge index. Hence, we redefine repetition path's destinations index (RPD-index) to reduce the redundancy of RP-index [5] in this section.

To construct the index scalably, we reform the structure of our index by changing the unit of indexing graph elements and compressing repetition paths being indexed. This study aims to enable GDBMSs to traverse repetition paths efficiently while judging whether or not a node is an HDN, as well as our previous study. Therefore, it is necessary to consider the labels of the edges that are often traversed repeatedly in an application to differentiate HDNs from all other nodes. Note that our approach requires one type of label for edges to collect while traversing repetition paths. Accordingly, this study set the repetition paths consisting of edges with just the selected label to make it easy enabling us to understand the meaning of traversing the paths.

Fig. 1 shows a schema of RPD-index structured with a hash map, whose keys are a pair of an HDN's ID and a length of repetition paths, and values are destinations of the paths. Differences in our approach compared with our previous study are as follows:

- A unit of RPD-index is changed to destination nodes (hereinafter we shortly called destinations) from repetition paths to be manageable in size, so our index is renamed to RPD-index. This is because almost all queries with repetition paths use destinations but paths.
- RPD-index's values store destinations, which contain three sets of categorized nodes by HDNs, LDNs, and operations performing recurrent index-scan.

[*1] This pattern means traversing a relationship repeatedly named `Relation` for two to four times.

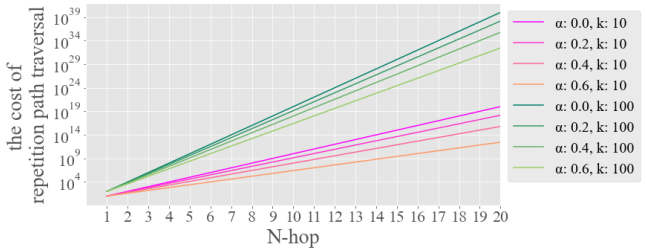

Fig. 2: The cost of repetition path traversal depends on nodes' degree (α: $\frac{\#\text{ of HDNs}}{\#\text{ of LDNs}}$, k: average degree of nodes).

- We innovate an operation enabling us to decompress omitted destinations by recurrently scanning RPD-index.

3.2 The cost of constructing RPD-index

In this section, we deal with the problem that an RP-index contains redundancy caused by duplicates of a part of the repetition path. The reason why the duplicates are included is that our indexing algorithm of RP-index likewise performs a traversal for other HDNs in the middle of repetition paths from an HDN of the index target. Moreover, this redundant process makes the consumed time for indexing repetition paths long; the time would be endless for a huge graph.

Traversing long repetition paths has a significant cost that depends on the degrees of the nodes in the paths. We can roughly estimate the cost of naïve repetition path traversal $E^{\mathrm{RP}}_{\mathrm{cost}}$ at n-hop with Formula (1):

$$E^{\mathrm{RP}}_{\mathrm{cost}} = k^n. \tag{1}$$

Thus, $E^{\mathrm{RP}}_{\mathrm{cost}}$ exponentially increases depending on the average degree k of nodes and the number n of hops.

Our approach for compressing an RPD-index introduces an operation of the recurrent scan, and it eliminates the cost of constructing redundant nodes being indexed. Given the average ratio α of the number of HDNs over that of LDNs in the neighbor nodes of each node, the cost of constructing RPD-index can be estimated as Formula (2):

$$\begin{aligned} E^{\mathrm{RPD}}_{\mathrm{cost}} &= k \times (1-\alpha)k \times \cdots \times (1-\alpha)k \\ &= (1-\alpha)^{n-1}k^n, \end{aligned} \tag{2}$$

where the first cost is just k exceptionally because the initial hop requires scanning all neighbor nodes.

Fig. 2 shows that the estimated cost of repetition path traversal of constructing RPD-index for different α and k, in which the x-axis denotes the number n of hops of a repetition path, and the y-axis represents the estimated costs by Formula (2) in logarithmic scale. In typical real-world networks following the power-law degree distribution, α ranges [0.0, 0.6] and k ranges [0, 100] depend on the size of the graph [1]. The figure indicates that omitting redundant traversals when reaching to HDNs in repetition paths can reduce the costs in a number of magnitudes, which can be seen from the comparison between $\alpha = 0.0$ (equivalent to $E^{\mathrm{RP}}_{\mathrm{cost}}$) to the other values of α.

3.3 HDNs selection

To construct RPD-index, a target edge label and a criterion of HDN need to be decided. After the choice of a target edge label, HDNs can be determined by the following steps:

(1) Extracting sub-graphs by scanning edges with the chosen label, then getting nodes connected with these edges,
(2) Calculating degree of each node,
(3) Creating degree distribution of whole extracted sub-graph described above,
(4) Determining threshold θ_{HDN} of node degrees for distinguishing between HDNs and LDNs.

From Step (1) to Step (3) are to draw degree distribution of each sub-graph constructed by target labeled edges. At the final step, though the thershold θ_{HDN} must be given, in practice, a heuristic approach called the Pareto law (a.k.a. 80:20 principle) [10] can be used, that is, nodes with the highest 20% degrees are determined as HDNs. This can be supported by the report by Barabási [11]. As a result, the time complexity of the above steps is $\mathcal{O}(n)$, which major cost is node degree calculation that depends on n.

3.4 An algorithm for constructing RPD-index

We describe an algorithm for constructing compressed RPD-index as shown in Algorithm 1 and 2. Algorithm 1 assumes that an input is a set of HDNs decided in Section 3.3. Algorithm 1 collects destinations of repetition paths from each HDN by invocating a function named CREATECOMPRESSEDRPDINDEX in Line 3.

Algorithm 2 defines CREATECOMPRESSEDRPDINDEX constructing RPD-index of a specified HDN. Indexing destinations of repetition paths begins from an inputted HDN, and a function TRAVERSE1HOPFROM performs to collect destinations of the first hop from the HDN. Indexing destinations of repetition paths continue until candidate nodes of the next traversal are nothing as written in Line 5 of Algorithm 2. As described in Section 3.1, destinations contain of HDNs, LDNs, and recurrent scan operations. Hence, Algorithm 2 indexes destinations while judging a type of a destination in "next" between Line 10 and 16. If a type of destination is an LDN, RPD-index appends the node as is, but in the case of otherwise our index generates or updates an operation performing recurrent index-scan, the operations are CREATERECURRENTSCAN and UPDATERECURRENTSCAN, respectively. Here, our algorithm does not perform TRAVERSE1HOPFROM for HDNs, then it realizes compression against the redundancy of RPD-index such as described in Section 3.2. When Algorithm 2 finishes traversing for the candidates, it returns an RPD-index for a specified HDN.

Finally, Algorithm 1 creates RPD-index for each HDN, which segregates the collected destinations of repetition paths by their lengths.

4. Experimental Evaluation

In order to evaluate whether our approach can construct

Algorithm 1 Collecting destinations from each HDN

Input H: a set of HDNs.
Output RPDi: RPD-index

```
RPDi ← {}
for each hdn ∈ H do
    RPDi[hdn] ← createCompressedRPDIndex(hdn)
end for
return RPDi
```

Algorithm 2 Constructing compressed RPD index

Input hdn: an HDN
Output hop2nodes: a hash map (hop $n \rightarrow$ destinations)

```
function indexCompressedRPD(next)
    hop ← 1
    next ← traverse1HopFrom(hdn)
    hop2nodes[hop] ← next
    while next.size() != 0 do
        current ← next
        next ← {}
        hop ← hop + 1
        for each c ∈ current do
            if c ∈ LDN then
                next.add(traverse1HopFrom(c))
            else if c ∈ HDN then
                next.add(createRecurrentScan(c))
            else if c ∈ RSO then
                next.add(updateRecurrentScan(c))
            end if
        end for
        hop2nodes[hop] ← next
    end while
    return hop2nodes
end function
```

Table 1: Our experimental environment

property	description
OS	CentOS 7.6.1801 (x86_64)
CPU	Intel Xeon Silver 4114 (2.20 GHz, 10 core) × 2
RAM	256 GB
Neo4j	ver. 4.0.0

Table 2: Datasets of LDBC SNB

SF	# of nodes	# of edges	# of properties
1	3,181,724	17,256,038	22,981,116
10	29,987,835	176,623,445	228,484,141
100	337,403,991	2,286,478,782	2,037,611,697

RPD-index faster, we conducted an experiment to compare our method for constructing compressed an RPD-index with a non-compressed one. We employed Neo4j version 4.0.0 [12] as a target for comparison because it exhibits the ability of high-speed graph traversing [13]. Our computational environment in our experiment is shown in Table 1.

4.1 Benchmark

For this experiment, we required vast and complex real graph data and its queries to evaluate the performance of traversing repetition paths and its scalability. Therefore, we utilize the social network benchmark (SNB) developed by Linked Data Benchmark Council (LDBC) [14] because it enables us to adjust the dataset volume.

The LDBC SNB can generate a dataset modeled as a social network and adjust data volume with a scale factor (SF). Hence, this benchmark enables us to evaluate the scalability of GDBMSs.

We performed the following tasks to prepare for this experiment.

- We prepared three sizes of LDBC SNB dataset by setting the SF values of 1, 10, and 100 to the program [15] as shown in Table 2,
- stored the datasets into Neo4j by using the program [16], and
- finally decided whether each node is an HDN or an LDN according to degree distribution as described in Section 3.3. In this experiment, we prepared three types of threshold θ_{HDN} at $\theta_{1\%}$, $\theta_{10\%}$, and $\theta_{20\%}$ to permit us to observe the trends of a consumed time.

The queries in the IC [17] include two kinds of repetition paths. The first kind consists of edges labeled relationship `knows`, representing an acquainted relationship between two users. In contrast, the second kind consists of edges labeled `replyOf`, which indicates a reply message to a post or a comment. In this experiment, we employ `knows` relationships for an indexing target because the pattern is many-to-many connection, and a repeated traversal becomes easily heavy cost.

4.2 The performance for constructing RPD-index

We assessed the effectiveness of our approach by constructing a compressed RPD-index in various data volumes, as shown in Table 2. In particular, we evaluate the consumed time for constructing a compressed RPD-index compared with a non-compressed one.

As described in Section 4.1, we conducted RPD-index construction for the `Person-knows-Person` pattern in this experiment. We measured a consumed time for constructing RPD-index and compressed one five times under each SF and threshold θ_{HDN}, and calculated each average of the times.

Fig. 3 shows the results comparing the consumed times for constructing RPD-index and compressed one. In order to visualize how many times faster constructing compressed an RPD-index compares with a non-compressed one, Fig.3 draws the ratio of the consumed time by the compressed approach compared with the naïve approach, on each length of a repetition path. Note that the naïve approach is a baseline in this experiment so its ratio of the consumed time is one in any length of a repetition path. Moreover, each x-axis in Fig. 3 denotes the length of repetition paths, and the value means the number n of n-hop from each HDN by indexing approaches. Furthermore, the compressed approach's processes absolutely finish indexing by shorter compared by naïve approaches. Hence, Fig. draws naïve approaches' results in accordance with the maximum length of compressed

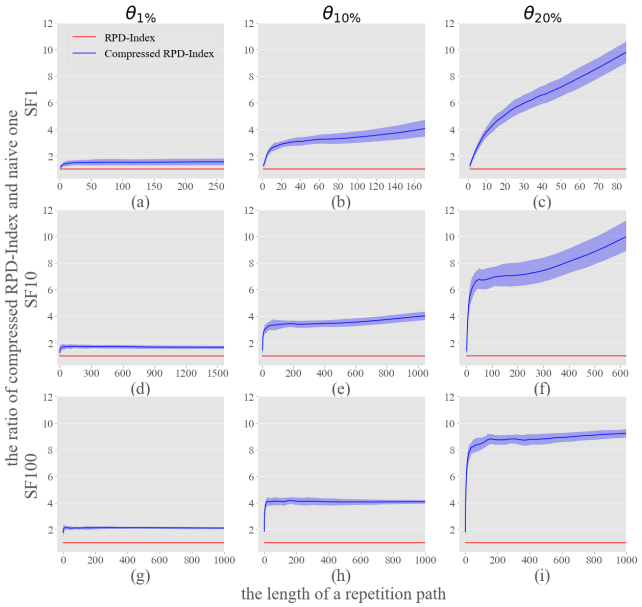

Fig. 3: Efficiency of compression. Compressed RPD-index can construct efficiently compared with the case of non-compressed one (i.e., our previous RP-index).

ones. In invisible results of the naïve approaches, the processes for indexing do not terminate for one-day under SF is 100. Finally, the lines show the average value of consumed time for indexing five times, and the bands aligned with the lines do the error range. The upper limit e_{upper} and lower one e_{lower} of the error range were calculated by our formula (3):

$$e_{\text{upper}} = \frac{\mu_r + \sigma_r}{\mu_c - \sigma_r}, \quad e_{\text{lower}} = \frac{\mu_r - \sigma_r}{\mu_c + \sigma_c}, \tag{3}$$

where μ_r and μ_c denote the average of consumed times by our naïve approach and the compressed approach, respectively, σ_r and σ_c do the standard deviation of the times. Therefore, the range of bands is $[r_\mu - e_{\text{lower}}, e_{\text{upper}} - r_\mu]$, where r_μ is the ratio of the average consumed time for indexing between two approaches.

According to Fig. 3, in the aspect of scale factors, our compressed approach was able to finish indexing destinations, but the speeds did not depend on SFs but were constant. On the contrary, with regard to the thresholds θ_{HDN}, it indicated the values influence the consumed time for indexing RPD-index as compared among Fig. 3(a), (b), and (c) or (d), Fig. 3(d), (e), and (f) for instance. As observed θ_{HDN} and scale factor simultaneously, the ratio of the consumed time indexing by our naïve approach and our compressed approach increased according to increases of both values.

The result described above does not indicate the higher θ_{HDN} is better, but the limitation for effectively reducing the consumed time was $\theta_{20\%}$. This is because a degree distribution generally follows power-law so that the top 80% of degree distribution from the lower degree is occupied by LDNs as described in Section 3.3.

5. Conclusion

In this paper, we proposed a method for compressing our index, enabling GDBMSs to obtain repetition paths' destinations. According to experiment results for variable sizes of a graph, our approach made it faster to construct a compressed RPD-index and acquire the property of scalability.

In the foreseeable future, we will extend the versatility of the RPD-index how by removing a constraint that indexing targets limit a single type of relationship. That is, the RPD-index will permit to index a combination of two or more types of relationships.

Acknowledgments This work is partially supported by JSPS KAKENHI Grant Numbers JP21H03555.

References

[1] Barabási, A.-L. and Pósfai, M.: *Network Science*, Cambridge University Press (2016).
[2] Rodriguez, M. A. and Neubauer, P.: Constructions from Dots and Lines, *Bulletin of the American Society for Information Science and Technology*, Vol. 36, No. 6, pp. 35–41 (online), DOI: 10.1002/bult.2010.1720360610 (2010).
[3] DB-Engines: Ranking of Graph DBMS, `https://db-engines.com/en/ranking/graph+dbms`, (accessed on 8th Aug. 2022.).
[4] Sakr, S. and Pardede, E.: *Graph Data Management: Techniques and Applications*, IGI Global (2011).
[5] Kusu, K. and Hatano, K.: A Hub-based Graph Management for Efficient Repetition Path Traversing, *2021 IEEE International Conference on Big Data and Smart Computing (BigComp)*, IEEE, pp. 188–191 (online), DOI: 10.1109/BigComp51126.2021.00043 (2021).
[6] Rodriguez, M. A. and Neubauer, P.: The Graph Traversal Pattern, *Graph Data Management: Techniques and Applications* (Sakr, S. and Pardede, E., eds.), IGI Global, chapter 2, pp. 29–46 (online), DOI: 10.4018/978-1-61350-053-8.ch002 (2012).
[7] Robinson, I., Webber, J. and Eifrem, E.: *Graph Databases*, O'Reilly Media, Inc. (2015).
[8] Sakr, S. and Al-Naymat, G.: The Overview of Graph Indexing and Querying Techniques, *Graph Data Management: Techniques and Applications* (Sakr, S. and Pardede, E., eds.), IGI Global, chapter 4, pp. 71–88 (online), DOI: 10.4018/978-1-61350-053-8.ch004 (2012).
[9] Newman, M.: *Networks: An Introduction*, Oxford University Press (2010).
[10] Pareto, V. F. D.: La courbe des revenus, *Cours d'Èconomie Politique vol.* (II), Librairie Droz, chapter (I), pp. 299–345 (1964).
[11] Barabási, A.-L. and Frangos, J.: *Linked: The New Science Of Networks Science Of Networks*, Perseus Books Group (2002).
[12] Neo4j, Inc.: Neo4j's HP, `https://neo4j.com/`, (accessed on 8th Aug. 2022.).
[13] Lissandrini, M., Brugnara, M. and Velegrakis, Y.: Beyond Macrobenchmarks: Microbenchmark-based Graph Database Evaluation, *Proceedings of the VLDB Endowment*, Vol. 12, No. 4, pp. 390–403 (online), DOI: 10.14778/3297753.3297759 (2018).
[14] LDBC: LDBC's HP, `http://ldbcouncil.org/`, (accessed on 8th Aug. 2022.).
[15] LDBC: ldbc/ldbc_snb_datagen.git, `https://github.com/ldbc/ldbc_snb_datagen`, (accessed on 8th Aug. 2022.).
[16] LDBC: ldbc_snb_implementations.git, `https://github.com/ldbc/ldbc_snb_implementations`, (accessed on 8th Aug. 2022.).
[17] Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.-D. and Boncz, P.: The LDBC Social Network Benchmark: Interactive Workload, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, ACM, pp. 619–630 (online), DOI: 10.1145/2723372.2742786 (2015).