JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 33, 143-158 (2017)
DOI: 10.6688/JISE.2017.33.1.9

ScaDiGraph: A MapReduce-Based Method for Solving Graph Problems

MOHAMMADHOSSEIN BARKHORDARI AND MAHDI NIAMANESH
Information and Communication Technology Research Center
Tehran, 1599616313 Iran
E-mail: {Barkhordari; Niamanesh}@ictrc.ac.ir

Graph data contain a large volume of information, which must be analysed. A method is required that can quickly process the information in distributed and scalable architectures. Single node methods are not suitable because they cannot process large amounts of information on their processors and memories. Problems encountered in distributed environments must also be addressed. These include graph division problems, algorithm division problems, exchange states among hardware nodes, and network traffic. In this paper, a scalable and distributable MapReduce-based method, ScaDiGraph, is proposed that makes hardware nodes independent. With this method, each hardware node can process a subgraph without any information from the other nodes. ScaDiGraph converts iterative graph problems, such as the All Pairs Shortest Path (APSP), pattern matching, and loop detection, into non-iterative problems and makes them suitable for the MapReduce architecture. In the present paper, ScaDiGraph is used to solve APSP, loop detection, and pattern matching problems, and the results are compared with those obtained with other MapReduce- and non-MapReduce-based methods. The results show that when applying ScaDiGraph on graph data, which causes simultaneous algorithm execution on each node, the execution time decreases in comparison with the prominent existing methods.

Keywords: graph, big data, MapReduce, APSP, loop detection, pattern matching

1. INTRODUCTION

Today, because of the high rate of information generated, innovative methods are necessary for information management. To process a large volume of information, scalable and distributable methods are used. In these methods, a large problem is broken down into smaller problems, and each problem is processed by a single node. Each node processes information and generates output. Finally, a node collects the outputs and generates the results. One of the most popular methods in this area is MapReduce [24]. However, it is not always possible to divide a problem into smaller problems and solve them separately. Moreover, data and computational relationships among the generated subproblems make it impractical to use MapReduce-like methods to solve problems in this way. Some problems are iterative. With such problems, it is necessary to continue processing until a predefined threshold or predetermined number of iterations is reached. Graph computation is one the most important types of iterative problems. The growth in graph data and the need to analyze the data means that methods are needed that can rapidly solve graph problems. Simple graph division cannot be used to solve graph problems because it requires information on all the nodes and edges.

Received August 29, 2015; revised January 14 & April 5 & June 21, 2016; accepted July 17, 2016.
Communicated by Jan-Jan Wu.

these methods usually suffer from the following problems:

- Graph division problems: This problem relates to how to divide graphs. Graph division problems are important problems in distributed systems that can cause inefficiencies in many of the proposed methods [36, 39].
- Data exchange problems: This issue is common to many of the proposed methods and relates to the exchange of graph data structure and vertex states over a network. Network latency problems and the large size of the subgraphs mean that many of the proposed solutions are ineffective [17, 20, 36, 40].
- Network traffic problems: Some methods create heavy traffic on a network. Part of this traffic is created by unwanted or uncontrolled objects. Other parts are usually the result of messages, graph data structure, and vertex states [36, 39].
- Iteration support problems: This type of problem is associated with intermediate results management. For some of the problems, such as graph problems, it is necessary to have results of the current iteration as input to the next iteration. Some of the proposed methods are not appropriate for iteration support and result in the inefficient use of hardware. MapReduce-based solutions are usually prone to such inefficiency [40, 41].
- Message storage and processing problems: Some methods use message-passing protocols. In large graphs, the messages sent by other nodes must be considered because each node must store and process a large number of messages. In many of the proposed methods, some of the nodes remain idle while waiting to process the messages from other nodes, which results in hardware inefficiencies [36, 39].
- Data locality problems: In this type of problem, nodes do not have all the information that they need. Therefore, their processes are dependent on data from other nodes [17, 20, 36, 39-41]. To the best of our knowledge, all the methods proposed thus far have data locality problems.

In this paper, ScaDiGraph, as a MapReduce based method, is proposed to solve graph problems, such as the All Pairs Shortest Path (APSP), loop detection and pattern matching. The ScaDiGraph method divides a graph into subgraphs, and metadata on these subgraphs are then stored. Each hardware node solves a subgraph problem, and the results of the hardware nodes are combined according to the graph division metadata. The proposed method utilizes MapReduce scalability.

Importantly, unlike other proposed methods for graph problems, with the proposed solution, no data (*e.g.*, graph data, graph states, messages) must be exchanged among the hardware nodes. Each node can separately solve a subgraph problem and transmit the results to the next layer of a Mapper(s). Another important feature of ScaDiGraph is that it converts iterative graph problems over hardware nodes to iterative problems on each node that can be solved separately, which enables them to be solved by MapReduce architectures.

The structure of this paper is as follows. Section 2 investigates the preliminaries of MapReduce and graph problems. In Section 3, the related work is discussed. Section 4 focuses on the proposed method. Section 5 presents the evaluation of the proposed method. Sections 6 and 7 provide the conclusions and the discussion, respectively.

2. PRELIMINARIES

In this part, MapReduce and solutions to graph problem methods are discussed.

2.1 MapReduce

In this section, studies related to MapReduce design are discussed. According to [19], a decomposable algorithm, partitionable data, and a sufficiently small data partition are the main characteristics required for the effective use of MapReduce. In [6], classic MapReduce was optimized to decrease the data transformation load. In the method described in [6], having a shared area for the information was considered. This type of design is suitable for solving problems, such as the *k-nn* and top *k* queries. In [16], MPI (message passing interface) was used for message passing in a MapReduce structure. The goal of that paper was to decrease the amount of data transferred in the MapReduce network. In [21], a method was developed for tackling the workloads in hierarchical MapReduce architectures. HadUP was presented in [22]. HadUP is a modified version of Hadoop and uses a deduplication-based snapshot differential algorithm (D-SD) and update propagation. Haloop [11] is another type of MapReduce structure suitable for iterative problems. iMapreduce [9] also supports iterative processes. In [2], HDFS (Hadoop file system) was substituted with a concurrency-optimized data storage layer based on the BlobSeer data management service. In [4], a model was presented to estimate the I/O behaviour of MapReduce applications. In [3], optimization over the MapReduce structure was divided into five groups. Fig. 1 shows these groups.

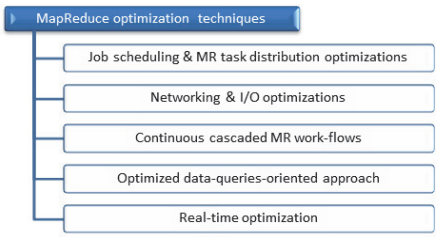

Fig. 1. MapReduce optimization techniques.

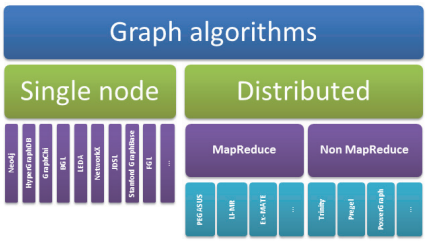

Fig. 2. Graph problem solutions classification.

2.2 Methods for Solving Graph Problems

There are several methods for solving graph problems. As shown in Fig. 2, solutions to graph problems can be divided into two categories: single node solutions and distributed solutions. Single node solutions, such as Neo4j [40], HyperGraphDB [34], FGL [27], JDSL [28], NetworkX [29], Stanford GraphBase [30], LEDA [31], and BGL [32], are usually not suitable for large graphs. Graphchi [37] is another single node solution for graph problems, but it solves subgraph problems sequentially and does not support synchronous task support.

In distributed solutions, there are two main categories: MapReduce-based methods and non-MapReduce-based methods. Almost all non-MapReduce methods are based on message passing. All the message-passing methods require space to catch the other nodes'

messages. In large graphs, the amount of space needed to solve graph problems could be too large. A large amount of message processing and increased network traffic are other important issues in large graphs. Message passing sometimes prevents the synchronous processing of nodes. Pregl [36] and GraphLab [35] are offline platforms used to solve graph problems; these do not support online queries. In [36], graph division is not transparent, which could have an adverse effect on the graph solution problems. PowerGraph [39] uses HDFS [42] to solve graph problems. This file system does not guarantee the best data locality. Unlike large graphs, states must be exchanged in subgraphs, which produces network traffic. Trinity [38] uses a file system similar to that employed by Hadoop [42], which does not support data locality for graph problems. Trinity is also a commercial system, and few details are available about the used methods. In [38], the graph division used to solve the graph problems is not clear. This is a major problem because synchronous computations depend on the division in the graphs.

3. RELATED STUDIES

According to the classification in [26], there are five groups of graph design patterns for MapReduce, as outlined in Fig. 3:

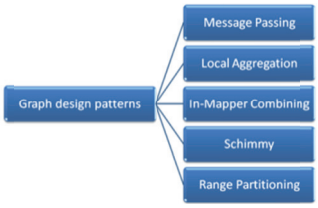

Fig. 3. Graph design patterns.

Message passing: In this pattern, each vertex using a local graph structure and vertex metadata are used to solve graph problems. The results are sent to neighbour vertices in an arbitrary message format. One of the main problems with this pattern is that the Mapper must send the graph structure and vertex states to neighbour vertices.

Local aggregation: In MapReduce methods, a large amount of information is moved among Mappers and Reducers. In the local aggregation pattern, Combiners are used to decrease information shuffling and traffic among Mappers and Reducers.

In-Mapper combining: The use of Combiners in practice is problematic. For example, it is not known how many times a Combiner is used or even whether it is used. Another problem is that although Combiners decrease the network traffic, they do not reduce the Key-Value pairs. Thus, many objects are constructed and deconstructed unnecessarily. According to the above descriptions, the In-Mapper combining design pattern is proposed. In this pattern, calculations on input records are performed on the Mapper side, and intermediate Key-Value pairs are not emitted until all the input records are processed.

Schimmy: In this pattern, unnecessary shuffling among Mappers and Reducers is omitted. Schimmy sorts graph vertices using a vertex key and then divides the graph into subgraphs. Then, it uses parallel merge join for parallel execution. It uses a Reducer for each subgraph, thereby decreasing the network traffic.

Range partitioning: In this pattern, a graph is divided into subgraphs with an equal number of vertices. Each Mapper processes the related subgraphs separately. The hash function is usually used for graph division. However, vertices and their related neighbours might not be included in a block. In practice, the properties of graphs are used to achieve effective partitions.

In [1], a method for executing a query over a distributed graph is proposed. MapReduce is employed in [5] to solve graph problems, such as graph transformation, subgraph partition, maximal clique enumeration, connected component finding, and community detection. In [7], a master worker method is used to solve graph problems of an iterative nature. A method is proposed in [8] for converting MapReduce jobs to bulk synchronous parallelism (BSP) programming model [10] jobs to utilize the BSO features for the graph computations. In [12], a combination of partial synchronization and locality enhancement is employed to alleviate synchronization overhead and achieve improved performance. In [13], a method for implementing recursive queries on a MapReduce structure is proposed to solve specific graph problems. The performance of MapReduce, join-side MapReduce, and BSP in solving different graph problems is compared in [15]. The authors concluded that BSP performs better with regard to iterative problems but that MapReduce is a better choice for enormous networks in which the structure cannot be fitted into the local machine memory. A distributed computing model is proposed in [17]. The model supports multi-iteration and random data access. In [18], MapReduce is utilized to find all the instances of a given sample graph in a larger data graph. High-level API is introduced in [20] for developing data-intensive applications for use in graph mining. A MapReduce implementation of an incremental APSP algorithm is developed in [23]. In [33], MapReduce is employed to solve different graph problems, such as PageRank. There are several problems with MapReduce-based methods. In some MapReduce-based methods, all or at least some parts of the graph structure must be sent through the network, which thereby increases the network traffic. Some MapReduce-based methods use Hadoop, which does not support best data locality for graph nodes. MapReduce does not support iterations, which is essential for many graph algorithms. Non-MapReduce methods have the following problems. Some non-MapReduce methods use message passing, which increases the network traffic. When using message-passing methods, the memory space required for processing messages must be considered. The amount of space required could increase dramatically in large graphs. Some of the methods use iterations. Each iteration does not finish until all the hardware nodes complete their tasks. This causes hardware usage inefficiency. Some of the methods prevent execution concurrency. No non-MapReduce methods account for data locality. To address the problems with the current MapReduce and non-MapReduce methods, we propose ScaDiGraph, which offers the following advantages. It can divide a graph into hardware nodes and execute an algorithm independently of individual nodes, without any data exchanges with other nodes. The graph can be divided into subgraphs that have equal size vertices. This division can be based on business knowledge or any arbitrary property. It uses locality and does not send the graph structure and status of the vertices over the network because all the required information is placed on the local hardware node. It can efficiently utilize many weak hardware nodes to solve large graph problems in a timely manner. Depending on the problem definition, ScaDiGraph can use distributable and scalable features of MapReduce architectures to improve the algorithm execu-

tion time and hardware performance. Some researchers [38] believe that MapReduce is not suitable for graph problems because of the iterative nature of the problems and the absence of an existing effective iteration management mechanism in MapReduce solutions. Although this concern is real, ScaDiGraph uses a method that omits inter-hardware node iterations, which makes graph problems suitable for MapReduce architectures.

4. PROPOSED METHOD

Because of the large amount of information that can be stored in a graph data structure and because of the importance of analysing graph information, it is essential to have a scalable and distributable framework for analysing the large amounts of information contained in a graph. Single-node solutions are not adequate for managing large graphs. ScaDiGraph omits iterations among hardware nodes by dividing a graph structure into smaller pieces. This type of division has two important merits. First, large graph iterative problems are divided into small, iterative problems, which can be solved by a hardware node. Second, many weak or medium hardware nodes can be used to solve large graph problems.

The proposed method divides the graph structure and solves problems on each hardware node separately. With the proposed architecture, all the results are gathered in the Reducer node. The hardware nodes act independently. In addition, graph data and states do not need to be exchanged, and the results of each hardware node are calculated separately. The following three steps describe the proposed method:

1. The main graph is divided into subgraphs; the number of subgraphs depends on the Mapper count. The same number of nodes can be used for each Mapper.
2. Each Mapper executes an algorithm on its subgraph.
3. The results of all Mappers plus the deleted edges are sent to the Reducer.

No paths are omitted in the second step, which means that the Mappers are non-filtering. Fig. 4 illustrates the architecture of the proposed method.

The operator for a combination of Mapper results is the following: Left or Right join. Table 1 shows the notations for this operator. In Table 1, <xy> is the edge added to the Mappers' results. When the calculation is complete, each Mapper node sends the results to the Reducer node with the format that appears in Table 2. The deleted edges are listed in the Reducer with the format shown in Table 3.

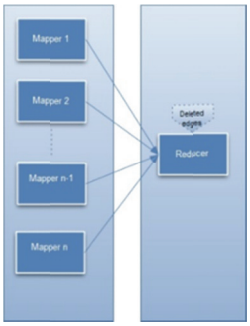

Fig. 4. Proposed method architecture.

Table 1. Notation for operator on mapper results.

Notation	Description
ϕ(<xy>,<zx>)	Left or Right join

Table 2. Mapper results format.

Source
Destination
Value

Table 3. Deleted edges format.

Source
Destination
Value
Visited

In the Reducer node, a database management system (DBMS) is used to achieve higher performance and a faster calculation speed in comparison with methods that do not use DBMSs. An in-memory database is used to decrease the I/O time on the hard disk drive (HDD). The table is partitioned to prevent a full scan of the table. In table design, partitioning is a method in which a table is partitioned by field values. In the query execution time, when there is a condition on the field and field value, only the related part of the table is scanned. When the Mappers send the results with the Table 2 format, partitioning must be applied to the "Source" and "Destination" fields at the same time. Because this arrangement is not possible on the same table, the Mapper results are stored in two tables: partitioning is applied to the "Source" field in one table and the "Destination" field in the other.

In this paper, ScaDiGraph is used to solve several large graph problems. The proposed method is applied to the all-pairs shortest paths (APSP), loop-detection, and pattern-matching problems. APSP is an algorithm that computes the shortest paths between all pairs of vertices. Loop-detection is another graph problem that is very important in solving various real problems. In this algorithm, all the loops in a graph are extracted and reported. Finally, pattern matching is an algorithm that detects a sequence of vertices in a graph.

4.1 APSP

APSP is an algorithm with $O(n3)$, where n is the number of vertices. Thus, with a large graph, it is impossible to solve a problem that contains many vertices in a timely manner. However, with ScaDiGraph, a large graph is divided into m (number of Mappers) small graphs. As a result, a large $O(n3)$ is converted to $O((n/m)3)$ smaller graphs, which enables the APSP algorithm to solve these graphical problems much faster. The following example illustrates the proposed method. In the case of the graph shown in Fig. 5, the input graph is divided into two subgraphs. There is no limitation on graph division. In this example, we use partitions that have an equal number of vertices. Fig. 6 illustrates the division of the graph. In this phase, each subgraph (X, Y) is assigned to a Mapper. Each Mapper calculates APSP for its subgraph. The results are shown in Table 4. In this stage, which is illustrated in Fig. 7, the deleted edges are added to the Mappers' results. New paths are calculated with a ϕ operator. The results from each node are sent to the Reducer. The results following the application of the join operators are shown in Table 5. The final results are shown in Table 6.

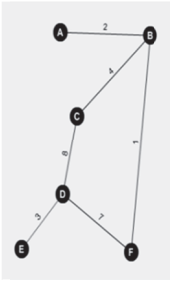

Fig. 5. Input graph.

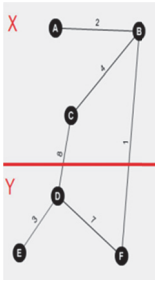

Fig. 6. Divided subgraph.

APSP algorithm (Psuedo code 1) is executed on each Mapper for the subgraphs and on the Reducer for the deleted edges. As shown in Psuedo code 1, all the paths for the graph nodes on the Mapper are calculated, and the results are stored in the Mapper_Edges_SRC table. Psuedo code 1 is also executed on the Reducer node for the nodes of the deleted edges, and the results are stored in the Deleted_Edges_SRC table. Psuedo code 2 is executed on the Reducer node for the APSP algorithm. As shown in Psuedo code 2, the following tasks are completed:

1. Mapper_Edges_SRC is copied to Mapper_Edges_DST to create partitions on the "Source" and "Destination" fields. Then, a local index is created on the partitions on the "Source" or "Destination" fields. If the partition is on the "Source" field, then the local index is created on the "Destination" field, and vice versa.
2. A while loop in the above code is repeated until there are unvisited rows in the Deleted_Edges_SRC table.
3. The "join" function is repeated up to the number of Deleted_Edges_SRC's partition count. The "join" function joins the related partitions of Mapper_Edges_SRC and Deleted_Edges_SRC and calculates the distances among the joined nodes.
4. After the "join" function, new edges that previously did not exist are added to Deleted_Edges_SRC to execute the related calculations for the new edges.
5. Steps 3 and 4 are repeated for Deleted_Edges_SRC and Mapper_Edges_DST. Finally, the visited rows from Deleted_Edges_SRC are deleted.
6. For the Deleted_Edges_DST table, steps 3-5 are executed with Mapper_Edges_DST and Mapper_Edges_SRC.
7. Finally, the shortest paths for the "Source" and the "Destination" fields are selected.

4.2 Pattern Matching

To detect a pattern or a sequence of vertices, we use a modified version of the APSP algorithm. In pattern matching, the visited vertices rather than the path costs are extracted. For example, to find the "BCDF" sequence in the graph in Fig. 6, the graph is first divided into two subgraphs, as shown in Fig. 7. Each Mapper then retrieves the paths that have a maximum length of four edges (input string length). Table 7 shows the results achieved with the Mappers. The deleted edges are then added to the graph, as depicted in Fig. 8. New paths with a maximum length of four edges with the ϕ operator are extracted. Table 8 shows the results. In Table 8, all the paths that have a length of more than four edges are discarded. Table 9 shows the results after the Reducer phase. The pattern-matching algorithm (Psuedo code 3) is executed on each Mapper for the subgraphs and on the Reducer for the deleted edges. Psuedo code 3 execution process is the same as that for the APSP Mapper code, except that the code path among the graph nodes is calculated instead of the distance among the nodes. The maximum length of the path is equal to the length of the Input string for pattern matching. Psuedo code 4 execution process is the same as that for the APSP code; however, the final step extracts the rows that have "Value" fields equal to the Input string's length.

4.3 Loop Detection

The proposed method can be used for loop detection in a graph; the pattern-match-

ing solution can be used to detect a loop in a graph. If, in the Reducer phase, the Keys are repeated, then we have a loop in a graph. For loop extraction, we can detect duplicate Keys that do not have more than two common vertices in their paths. For example, to find the loops in the graph in Fig. 6, the graph is first divided into two subgraphs, as shown in Fig. 7. Each Mapper then extracts the paths. Table 10 presents the results of the Mappers. The deleted edges are then added to the graph, as demonstrated in Fig. 8. New paths with the ϕ operator are extracted. Table 11 shows the results. Because we have duplicated Keys in Table 11, we could have loop(s) in the graph. For loop extraction, the duplicated Keys are first identified. Paths that have the same Keys and less than three common vertices are then extracted as loops. The loop-detection algorithm is executed on each Mapper for the subgraphs and on the Reducer for the deleted edges. Psuedo code 5 execution process is the same as the pattern-matching process. The maximum length of the path is equal to the length of the input value for the loop detection. Psuedo code 6 is executed on the Reducer for the loop-detection algorithm. The above code-execution process is similar to that for pattern matching; however, at the final step, paths that do not have more than two nodes in common are extracted.

5. EVALUATION

The evaluation was divided into three parts: Implementation and evaluation of APSP, Pattern matching and Loop detection

5.1 APSP

To evaluate the proposed method, the APSP algorithm with the proposed method was applied to information in a social network composed of expert users. This social network has approximately 50,000 users, and each user is displayed as a graph node. The relationships among these users make up approximately 10,000,000 edges. Table 13 shows the specifications of the hardware nodes used for the proposed evaluation method. Each Mapper hardware node has 1,000 graph nodes. We divided the graph into subgraphs that have an equal number of nodes. Table 13 shows the specifications of the hardware nodes.

We divided the graph into subgraphs that have an equal number of nodes. The nodes can be loaded on servers in an arbitrary manner. Table 14 shows the input data format. We divided the main graph based on the ID field, which is the primary key of the input data. The ID field grows sequentially. Based on the maximum value of the ID field and the number of Mappers (M), we determine the lower bound and upper bound of the IDs, which are present on each node.

$$Mapper1\ (0 < ID <= Max(ID)/M); Mapper2\ (Max(ID)/M < ID <= 2 \times Max(ID)/M); \ldots; MapperM\ ((M-1) \times Max(ID)/M < ID <= Max(ID))$$

We used Redis [25] as the In-Memory DBMS. Table 14 shows the results.

5.2 Pattern Matching

For pattern matching, we used bank transactions to detect any suspicious transaction

Table 4. Calculated APSP on each subgraph.

Subgraph	Key	Value
X	AB	2
	BC	4
	AC	6
Y	DE	3
	DF	7
	EF	10

Table 5. Mapper results.

Operator and Key	Key	Value
-	AB	2
-	BC	4
-	AC	6
-	DE	3
-	DF	7
-	EF	10
ϕ(<AB>,<BF>)	AF	3
ϕ(<BC>,<CD>)	BD	12
ϕ(<AC>,<CD>)	AD	14
ϕ(<CD>,<DE>)	CE	11
ϕ(<BF>,<DF>)	CF	8
ϕ(<BF>,<EF>)	BE	11
ϕ(<DF>,<CD>)	CF	15
ϕ(<AF>,<DF>)	AD	10
ϕ(<AF>,<EF>)	AE	13
ϕ(<BD>,<DE>)	BE	15
ϕ(<BD>,<DF>)	BF	19
ϕ(<AD>,<DF>)	AF	21
ϕ(<AD>,<DE>)	AE	17
ϕ(<CF>,<DF>)	CD	12
ϕ(<CF>,<EF>)	CE	15
ϕ(<BD>,<BF>)	DF	13
ϕ(<CD>,<BD>)	BC	16

Table 6. Reducer results.

Key	Value
AB	2
AC	6
AD	10
AE	13
AF	3
BC	4
BD	12
BE	11
BF	19
CD	12
CE	11
CF	8
DE	3
DF	7
EF	10

Table 7. Extracted paths on each subgraph.

Subgraph	Key	Value
X	AB	AB
	BC	BC
	AC	ABC
Y	DE	DE
	DF	DF
	EF	EDF

Table 8. Mapper results.

Operator and Key	Key	Value
-	AB	AB
-	BC	BC
-	AC	ABC
-	DE	DE
-	DF	DF
-	EF	EDF
ϕ(<AB>,<BF>)	AF	ABF
ϕ(<BC>,<CD>)	BD	BCD
ϕ(<AC>,<CD>)	AD	ABCD
ϕ(<CD>,<DE>)	CE	CDE
ϕ(<BF>,<DF>)	BD	BFD
ϕ(<BF>,<EF>)	BE	BFDE
ϕ(<DF>,<CD>)	CF	CDF
ϕ(<AF>,<DF>)	AD	ABFD
ϕ(<AF>,<EF>)	AE	-
ϕ(<BC>,<CE>)	BE	BCDE
ϕ(<BD>,<DF>)	BF	BCDF
ϕ(<AD>,<DF>)	AF	-
ϕ(<AD>,<DE>)	AE	-
ϕ(<CF>,<DF>)	CD	CBFD
ϕ(<CF>,<EF>)	CE	-
ϕ(<DB>,<BF>)	DF	DCBF
ϕ(<CF>,<BF>)	BC	CDFB

Table 9. Reducer results.

Key	Value
BF	BCDF
CD	CBFD
DF	DCBF
BC	CDFB

Fig. 7. Adding deleted edges.

Table 10. Extracted paths on each subgraph.

Subgraph	Key	Value
X	AB	AB
	BC	BC
	AC	ABC
Y	DE	DE
	DF	DF
	EF	EDF

Table 11. Mapper results.

Operator and Key	Key	Value
-	AB	AB
-	BC	BC
-	AC	ABC
-	DE	DE
-	DF	DF
-	EF	EDF
ϕ(<AB>,<BF>)	AF	ABF
ϕ(<BC>,<CD>)	BD	BCD
ϕ(<AC>,<CD>)	AD	ABCD
ϕ(<CD>,<DE>)	CE	CDE
ϕ(<BF>,<DF>)	CF	BFD
ϕ(<BF>,<EF>)	BE	BFDE
ϕ(<DF>,<CD>)	CF	CDF
ϕ(<AF>,<DF>)	AD	ABFD
ϕ(<AF>,<EF>)	AE	ABFDE
ϕ(<BD>,<DE>)	BE	BCDE
ϕ(<BD>,<DF>)	BF	BCDF
ϕ(<AD>,<DF>)	AF	ABCDF
ϕ(<AD>,<DE>)	AE	ABCDE
ϕ(<CF>,<DF>)	CD	CBFD
ϕ(<CF>,<EF>)	CE	CBFDE
ϕ(<BD>,<BF>)	DF	DCBF
ϕ(<CF>,<BF>)	BC	CDFB

Table 12. Reducer results.

Key	Value
AD	~~ABCD~~
AD	~~ABFD~~
AE	~~ABCDE~~
AE	~~ABFDE~~
AF	~~ABCDF~~
AF	~~ABF~~
BC	BC
BC	CDFB
BE	~~BCDE~~
BE	~~BFDE~~
CE	~~CBFDE~~
CE	~~CDE~~
CF	BFD
CF	CDF
DF	DCBF
DF	DF

```
For k=1 to SubgraphNodesCount
   For j=1 to SubgraphNodesCount
      For i=1 to SubgraphNodesCount
         {
            If  (dis-
tance(node(k), node(i)) != infin-
ity) && (distance(node(i),
node(j)) != infinity )
               {
                  distance(node(k),
node(j))= distance(node(k),
node(i)) + distance(node(i),
node(j));
                  Insert into Map-
per Results(Source,Destination,Val
ue)
                           Val-
ues(node(k). Name, node(j). Name,
distance(node(k), node(j)));
               }
         }
Send Edges(Mapper Results,Reducer.
Mapper_Edges_SRC);
```

Pusedo code 1

```
For k=1 to SubgraphNodesCount
   For j=1 to SubgraphNodesCount
      For i=1 to SubgraphNodesCount
         {
            If  (length(path(node(k),
node(i))) <= length(InputPattern)) &&
   (length(path(node(i), node(j))) <=
            length(InputPattern))

               {
                  path(node(k), node(j))=
path(node(k), node(i)) + path(node(i),
node(j));
                  if (length(path(node(k),
node(j)))<= length(InputPattern))
                     {
                        Insert into Map-
per Results(Source,Destination,Value)
                                 Values(node(k).
Name, node(j). Name, path(node(k),
node(j)));
                     }
               }
         }
Send Edges(Mapper Results,Reducer.
Mapper_Edges_SRC);
```

Pusedo code 3

```
For k=1 to SubgraphNodesCount
   For j=1 to SubgraphNodesCount
      For i=1 to SubgraphNodesCount
         {
            If(length(path(node(k),
node(i))) <= InputLength) &&
   (length(path(node(i), node(j))) <=
            InputLength)

               {
                  path(node(k), node(j))=
path(node(k), node(i)) + path(node(i),
node(j));
                  if (length(path(node(k),
node(j)))<= InputLength)
                     {
                        Insert into Map-
per Results(Source,Destination,Value)
                                 Values(node(k).
Name, node(j). Name, path(node(k),
node(j)));
                     }
               }
         }
Send Edges(Mapper Results,Reducer.
Mapper_Edges_SRC);
```

Pusedo code 5

```
Copy Edges(Mapper Edges SRC, Map-
per Edges DST);
Copy_Edges(Deleted_Edges_SRC, Delet-
ed_Edges_DST);
Create local index on column "Destination"
for each partition on table "Map-
per Edges SRC";
Create local index on column "Source" for
each partition on table "Mapper Edges DST";
While (Select exists (visited) from
Deleted_Edges_SRC where visited=0){
For i= 1 to Deleted_Edges_SRC. Parti-
tionCount {
Join(Mapper_Edges_SRC. parti-
tion(Deleted Edges SRC. Partition(i).
PartitionName),       Deleted Edges SRC.
partition (Deleted Edges SRC. Parti-
tion(i). PartitionName));
Deleted Edges SRC. Partition(i). visit-
ed=1;
Insert join results into Mapper_Edges_SRC;
Insert join results into Mapper_Edges_DST;
Insert new edges into Deleted_Edges_SRC;
Insert new edges into Deleted Edges DST;
Join(Mapper Edges DST. parti-
tion(Deleted Edges SRC. Partition(i).
PartitionName),       Deleted Edges SRC.
partition (Deleted Edges SRC. Parti-
tion(i). PartitionName));
Insert join results into Mapper_Edges_SRC;
Insert join results into Mapper_Edges_DST;
Insert new edges into Deleted Edges SRC;
Insert new edges into Deleted Edges DST;
Delete from Deleted Edges SRC where
visited=1; }
 For i= 1 to Deleted Edges DST. Parti-
tionCount {
Join(Mapper Edges SRC. parti-
tion(Deleted_Edges_DST. Partition(i).
PartitionName),       Deleted_Edges_DST.
partition (Deleted_Edges_DST. Parti-
tion(i). PartitionName));
Deleted Edges DST. Partition(i). visit-
ed=1;
Insert join results into Mapper Edges SRC;
Insert join results into Mapper Edges DST;
Insert new edges into Deleted Edges SRC;
Insert new edges into Deleted Edges DST;
Join(Mapper_Edges_DST. parti-
tion(Deleted_Edges_DST. Partition(i).
PartitionName),       Deleted Edges DST.
partition (Deleted Edges DST. Parti-
tion(i). PartitionName));
Insert join results into Mapper Edges SRC;
Insert join results into Mapper Edges DST;
Insert new edges into Deleted Edges SRC;
Insert new edges into Deleted Edges DST;
Delete from Deleted_Edges_DST where
visited=1; } }
Select Source, Destination, Min(Value) From
Mapper Edges SRC group by Source, Desti-
nation;
```

Pusedo code 2

```
Copy Edges(Mapper Edges SRC, Map-
per Edges DST);
Copy_Edges(Deleted_Edges_SRC, Delet-
ed_Edges_DST);
Create local index on column "Destination"
for each partition on table "Map-
per Edges SRC";
Create local index on column "Source" for
each partition on table "Mapper Edges DST";
While (Select exists(visited) from De-
leted_Edges_SRC where visisted=0)
{For i= 1 to Deleted_Edges_SRC. Parti-
tionCount
 {Join(Mapper_Edges_SRC. parti-
tion(Deleted Edges SRC. Partition(i).
PartitionName),       Deleted Edges SRC.
partition (Deleted Edges SRC. Parti-
tion(i). PartitionName)) ;
Deleted Edges SRC. Partition(i). visit-
ed=1;
Filter results where length(results. path)
< length(InputPattern));
Insert join results into Mapper_Edges_SRC;
Insert join results into Mapper Edges DST;
Insert new edges into Deleted Edges SRC;
Insert new edges into Deleted Edges DST;
Join(Mapper Edges DST. parti-
tion(Deleted Edges SRC. Partition(i).
PartitionName),       Deleted Edges SRC.
partition (Deleted_Edges_SRC. Parti-
tion(i). PartitionName)) ;
Filter results where length(results. path)
<length(InputPattern));
Insert join results into Mapper Edges SRC;
Insert join results into Mapper Edges DST;
Insert new edges into Deleted Edges SRC;
Insert new edges into Deleted Edges DST;
Delete from Deleted Edges SRC where
visited=1; }
 For i= 1 to Deleted_Edges_DST. Parti-
tionCount
 {Join(Mapper Edges SRC. parti-
tion(Deleted Edges DST. Partition(i).
PartitionName),       Deleted Edges DST.
partition (Deleted Edges DST. Parti-
tion(i). PartitionName)) ;
Deleted Edges DST. Partition(i). visit-
ed=1;
Filter results where length(results.
path)<length(InputPattern));
Insert join results into Mapper Edges SRC;
Insert join results into Mapper Edges DST;
Insert new edges into Deleted Edges SRC;
Insert new edges into Deleted Edges DST;
Join(Mapper Edges DST. parti-
tion(Deleted Edges DST. Partition(i).
PartitionName),       Deleted Edges DST.
partition (Deleted_Edges_DST. Parti-
tion(i). PartitionName)) ;
Filter results where length(results.
path)<length(InputPattern));
Insert join results into Mapper Edges SRC;
Insert join results into Mapper Edges DST;
Insert new edges into Deleted Edges SRC;
Insert new edges into Deleted Edges DST;
Delete from Deleted Edges DST where
visited=1; } }
Select Source, Destination, Value From
Mapper_Edges_SRC where Value= InputPat-
tern;
```

Pusedo code 4

```
Copy Edges(Mapper Edges SRC, Map-
per Edges DST);
Copy_Edges(Deleted_Edges_SRC, Delet-
ed_Edges_DST);
Create local index on column "Destination"
for each partition on table "Map-
per Edges SRC";
Create local index on column "Source" for
each partition on table "Mapper Edges DST";
 While (Select exists(visited) from
Deleted_Edges_SRC where visisted=0)
{For i= 1 to Deleted_Edges_SRC. Parti-
tionCount {
Join(Mapper_Edges_SRC. parti-
tion(Deleted Edges SRC. Partition(i).
PartitionName),       Deleted Edges SRC.
partition (Deleted Edges SRC. Parti-
tion(i). PartitionName)) ;
Deleted Edges SRC. Partition(i). visit-
ed=1;
Filter results where length(results.
path)<= InputLength;
Insert join results into Mapper_Edges_SRC;
Insert join results into Mapper Edges DST;
Insert new edges into Deleted Edges SRC;
Insert new edges into Deleted Edges DST;
Join(Mapper Edges DST. parti-
tion(Deleted Edges SRC. Partition(i).
PartitionName),       Deleted Edges SRC.
partition (Deleted_Edges_SRC. Parti-
tion(i). PartitionName)) ;
Filter results where length(results.
path)<= InputLength;
Insert join results into Mapper Edges SRC;
Insert join results into Mapper Edges DST;
Insert new edges into Deleted Edges SRC;
Insert new edges into Deleted Edges DST;
Delete from Deleted Edges SRC where
visited=1; }
 For i= 1 to Deleted_Edges_DST. Parti-
tionCount {
Join(Mapper Edges SRC. parti-
tion(Deleted Edges DST. Partition(i).
PartitionName),       Deleted Edges DST.
partition (Deleted Edges DST. Parti-
tion(i). PartitionName)) ;
Deleted Edges DST. Partition(i). visit-
ed=1;
Filter results where length(results.
path)<= InputLength;
Insert join results into Mapper Edges SRC;
Insert join results into Mapper Edges DST;
Insert new edges into Deleted Edges SRC;
Insert new edges into Deleted Edges DST;
Join(Mapper Edges DST. parti-
tion(Deleted Edges DST. Partition(i).
PartitionName),       Deleted Edges DST.
partition (Deleted_Edges_DST. Parti-
tion(i). PartitionName)) ;
Filter results where length(results.
path)<= InputLength;
Insert join results into Mapper Edges SRC;
Insert join results into Mapper Edges DST;
Insert new edges into Deleted Edges SRC;
Insert new edges into Deleted Edges DST;
Delete from Deleted Edges DST where
visited=1; } }
 Duplicate_Set =(Select Source, Destina-
tion, Value From Mapper_Edges_SRC having
count(concat(Source, Destination))>1)
For each concat(Source, Destination) in
Duplicate Set
{   For i= 1 to Count(Values)-1 //Number of
paths
    For j= i+1 to Count(Values)     {
       If (Inter-
sect(Value(i),Value(j))==2)    Insert into
Results(Value(i),Value(j))    }  }
```

Pusedo code 6

sequences among the customers. The customers and their transactions are considered to be graph nodes and edges, respectively. We found patterns that have a length of ten edges. This bank has approximately 130,000 customers and approximately 2,500,000 transactions in three months. The results are shown in Table 16.

5.3 APSP

For loop detection, we used the bank transactions described in section 5.2 to find

loops that have a length of ten edges. The detection of loops among the customers signifies fraudulent activity, such as money laundering or fake transactions aimed at falsely increasing the turnover. The results are shown in Table 17. The detected loops were sent to the bank's fraud-detection office for further investigation.

5.4 Comparison of the Proposed Method with Other Methods

We applied the Pegasus, Pregel (Graph 1.0.0), and Power Graph 2.2 algorithms to graphs on fifty servers using the specifications shown in Table 18. The total RAM memory used for fifty nodes in ScaDiGraph is 912 GB, and the total HDD used is 10.5 TB. Fifty core i5 CPUs and a core i7 CPU are used as the processors. The total RAM used for each of the other methods (Pegasus, Pregel, and Power Graph) is 3.2 TB, and the total HDD used is 50 TB. Fifty core i7 CPUs are used as processors. Thus, it can be seen that ScaDiGraph uses less hardware resources (RAM, HDD and CPU) to solve graph problems. The results are shown in Fig. 8. The detected loops were sent to the bank's fraud-detection office for further investigation.

6. DISCUSSION

The proposed method works better than Pregl, Pegasus and Power graphs because all these methods must exchange intermediate results among the hardware nodes. Exchanging messages among the hardware nodes has two main problems. The first problem is that each hardware node must store messages from other hardware nodes. If we have a large graph, then we have too many subgraphs, and therefore, we require a large amount of memory to maintain and process the messages and we must have a message process queue. On the other hand, message exchange among hardware nodes causes network congestion. Both network congestion and message process queues cause the sender hardware nodes to wait, which causes improper use of the processing power and memory capacity. Nevertheless, in ScaDiGraph, there is no relation among the Mapper hardware nodes. Each Mapper works with its subgraph, and therefore, data locality is completely met. In other words, all the data necessary to execute an algorithm on the hardware node are located on the same hardware node; therefore, we have avoided message exchange among the hardware nodes and its consequences.

The best execution time for ScaDiGraph occurs when we have an isolated subgraph on each Mapper (thus, there is no deleted edge and no calculation on the Reducer node). Fig. 9 shows the best case. The worst execution time occurs when all the nodes on each Mapper have no relation with other nodes and all the relations among the nodes are added at the deleted edges. In such cases, ScaDiGraph cannot improve the execution time, and all the calculations must be performed on the Reducer node. Fig. 10 shows the worst case. We used two techniques to solve the graph problems separately on each hardware node. First, we unified the data format on each hardware node. Second, we changed the algorithm in such a way that each node can solve its problem in a solitary way. We have used this technique to solve problems in other fields, such as data mining [40] and databases [41], and this technique has achieved lower execution times than other prominent existing methods.

Table 13. Hardware node specifications.

Mapper virtual hardware nodes	CPU	Intel Core i5-6500T Processor (6M Cache, up to 3.10 GHz)
	HDD	10 GB
	RAM	8 GB
Reducer hardware nodes	CPU	Intel Core i7-6700T Processor (8M Cache, up to 3.60 GHz)
	HDD	10 TB
	RAM	512 GB

Table 14. Input data format.

ID
Source
Destination
Value

Table 18. Hardware node specifications.

Hardware nodes	CPU	Intel Core i7-6700T Processor (8M Cache, up to 3.60 GHz)
	HDD	10 TB
	RAM	64 GB

Table 15. ScaDiGraph execution time for APSP.

Stage name	Time(minute)
Mapper	14
Reducer	301

Table 16. Execution time for pattern matching.

Stage name	Time(minute)
Mapper	12
Reducer	284

Table 17. Execution time for loop detection.

Stage name	Time(minute)
Mapper	12
Reducer	310

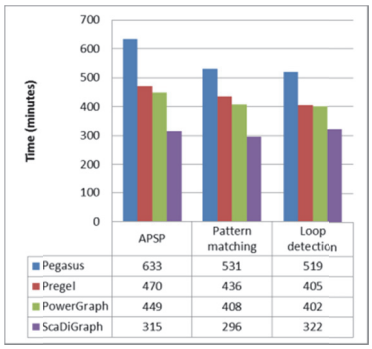

Fig. 8. Comparing ScaDiGraph with other methods.

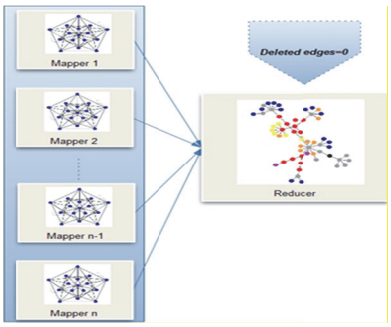

Fig. 9. ScaDiGraph best case.

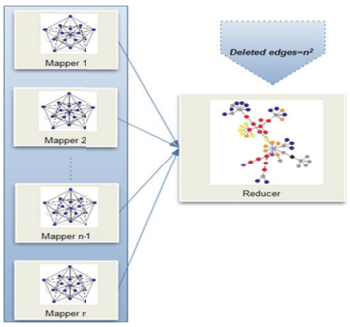

Fig. 10. ScaDiGraph worst case.

7. CONCLUSIONS

The large amount of information in graph data structures requires scalable and distributable information processing methods. Many scalable methods have been proposed to solve large graph problems, but some of these create heavy traffic on the network because of non-consideration of data locality. Some methods use iterations to solve graph problems. However, in these methods, the next iteration cannot be started until all the previous iteration tasks have been completed, which gives rise to hardware inefficiency.

In this paper, we introduced a method, ScaDiGraph, which is based on MapReduce, to solve graph problems, such as APSP, pattern matching and loop detection. ScaDi-

Graph divides a large graph into subgraphs. Each node of the subgraph executes an algorithm without the need for information about other subgraphs. By converting large graph problems into subgraphs, the proposed method can solve graph problems in a timely manner. Another advantage of the proposed method is that commodity hardware nodes can be used to solve large graph problems. By converting the iterative nature of graph problems into non-iterative problems, ScaDiGraph makes it possible to solve these problems using MapReduce methods. The method was applied to two case studies: an expert social network for which APSP algorithms were used and a bank's transactions for which pattern matching and loop detection problems were solved.

REFERENCES

1. M. Sarwat, *et al.*, "Horton: Online query execution engine for large distributed graphs," in *Proceedings of IEEE 28th International Conference on Data Engineering*, 2012, pp. 1289-1292.
2. B. Nicolae, *et al.*, "BlobSeer: Bringing high throughput under heavy concurrency to Hadoop map-reduce applications," in *Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, 2010, pp. 1-11.
3. A. Osman, E.-R. Amr, and A. Elnaggar, "Towards real-time analytics in the cloud," in *Proceedings of IEEE 9th World Congress on Services*, 2013, pp. 428-435.
4. S. G root, "Modeling i/o interference in data intensive map-reduce applications," in *Proceedings of IEEE/IPSJ 12th International Symposium on Applications and the Internet*, 2012, pp. 206-209.
5. S. Yang, *et al.*, "Efficient dense structure mining using mapreduce," in *Proceedings of IEEE International Conference on Data Mining Workshops*, 2009, pp. 332-337.
6. L. Ding, *et al.*, "Commapreduce: An improvement of mapreduce with lightweight communication mechanisms," in *Proceedings of International Conference on Database Systems for Advanced Applications*, 2012, pp. 150-168.
7. R. Lichtenwalter and N. V. Chawla, "DisNet: A framework for distributed graph computation," in *Proceedings of IEEE International Conference on Advances in Social Networks Analysis and Mining*, 2011, pp. 263-270.
8. L. Fegaras, "Supporting bulk synchronous parallelism in map-reduce queries," in *Proceedings of IEEE High Performance Computing*, *Networking*, *Storage and Analysis Companion*, 2012, pp. 1068-1077.
9. Y. Zhang, *et al.*, "Imapreduce: A distributed computing framework for iterative computation," *Journal of Grid Computing*, Vol. 10, 2012, pp. 47-68.
10. L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, Vol. 33, 1990, pp. 103-111.
11. Y. Bu, *et al.*, "HaLoop: efficient iterative data processing on large clusters," in *Proceedings of the VLDB Endowment*, Vol. 3, 2010, pp. 285-296.
12. K. Kambatla, *et al.*, "Asynchronous algorithms in MapReduce," in *Proceedings of IEEE International Conference on Cluster Computing*, 2010, pp. 245-254.
13. F. N. Afrati, *et al.*, "Map-reduce extensions and recursive queries," in *Proceedings of the 14th ACM International Conference on Extending Database Technology*, 2011, pp. 1-8.
14. Message passing interface, http://www.mpi-forum.org/.

15. T. Kajdanowicz, K. Przemyslaw, and I. Wojciech, "Parallel processing of large graphs," *Future Generation Computer Systems*, Vol. 32, 2014, pp. 324-337.
16. H. Mohamed and M.-M. Stéphane, "MRO-MPI: MapReduce overlapping using MPI and an optimized data exchange policy," *Parallel Computing*, Vol. 39, 2013, pp. 851-866.
17. Q. Li, *et al.*, "LI-MR: a local iteration map/reduce model and its application to mine community structure in large-scale networks," in *Proceedings of the 11th IEEE International Conference on Data Mining Workshops*, 2011, pp. 174-179.
18. F. N. Afrati, F. Dimitris, and J. D. Ullman, "Enumerating subgraph instances using map-reduce," in *Proceedings of the 29th IEEE International Conference on Data Engineering*, 2013, pp. 62-73.
19. F. Highland and J. Stephenson, "Fitting the problem to the paradigm: algorithm characteristics required for effective use of MapReduce," *Procedia Computer Science*, Vol. 12, 2012, pp. 212-217.
20. W. Jiang and G. Agrawal, "Ex-mate: data intensive computing with large reduction objects and its application to graph mining," in *Proceedings of the 11th IEEE/ACM International Symposium on Cluster*, *Cloud and Grid Computing*, 2011, pp. 475-484.
21. V. S. Martha, W. Zhao, and X. Xu, "*h*-MapReduce: a framework for workload balancing in MapReduce," in *Proceedings of the 27th IEEE International Conference on Advanced Information Networking and Applications*, 2013, pp. 637-644.
22. D. Lee, J.-S. Kim, and S. Maeng, "Large-scale incremental processing with MapReduce," *Future Generation Computer Systems*, Vol. 36, 2014, pp. 66-79.
23. S. S. Khopkar, N. Rakesh, and A. G. Nikolaev, "An efficient map-reduce algorithm for the incremental computation of all-pairs shortest paths in social networks," in *Proceedings of IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2012, pp. 1144-1148.
24. J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, Vol. 51, 2008, pp. 107-113.
25. Redis In-Memory database, http://redis. io/.
26. J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in MapReduce," in *Proceedings of the 8th ACM Workshop on Mining and Learning with Graphs*, 2010, pp. 78-85.
27. M. Erwig, "Inductive graphs and functional graph algorithms," *Journal of Functional Programming*, Vol. 11, 2001, pp. 467-492.
28. M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, John Wiley & Sons, 2008.
29. J. L. Gross and J. Yellen, *Graph Theory and its Applications*, Chapman and Hall/ CRC, 2005.
30. D. E. Knuth, *The Stanford GraphBase: A Platform for Combinatorial Computing*, Vol. 37, Addison-Wesley, NY, 1993.
31. K. Mehlhorn, S. Näher, and C. Uhrig, "The LEDA platform of combinatorial and geometric computing, 1999," http://dx. doi.org/10, 1007/3-540-63165-8 161: 7-16.
32. A. Lumsdaine, L. Q. Lee, and J. G. Siek, *The Boost Graph Library: User Guide and Reference Manual*, Addison-Wesley, MA, 2002.
33. U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proceedings of the 9th IEEE Interna-*

tional Conference on Data Mining, 2009, pp. 229-238.
34. H. Higaki, *et al.*, "Checkpoint and rollback in asynchronous distributed systems," in *Proceedings of the 16th Annual Joint Conference of the IEEE Computer and Communications Societies*, *Driving the Information Revolution*, Vol. 3, 1997, pp. 998-1005.
35. Y. Low, *et al.*, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," in *Proceedings of the VLDB Endowment 5*, Vol. 8, 2012, pp. 716-727.
36. G. Malewicz, *et al.*, "Pregel: a system for large-scale graph processing," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2010, pp. 135-146.
37. A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: large-scale graph computation on just a PC," Presented as part of *the 10th USENIX Symposium on Operating Systems Design and Implementation*, 2012.
38. B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2013, pp. 505-516.
39. J. E. Gonzalez, *et al.*, "Powergraph: Distributed graph-parallel computation on natural graphs," Presented as part of *the 10th USENIX Symposium on Operating Systems Design and Implementation*, 2012, pp. 17-30.
40. M. Barkhordari and M. Niamanesh, "ScadiBino: An effective MapReduce-based association rule mining method," in *Proceedings of the 16th ACM International Conference on Electronic Commerce*, 2014, p. 1.
41. M. Barkhordari and M. Niamanesh, "ScaDiPaSi: an effective scalable and distributable MapReduce-based method to find patient similarity on huge healthcare networks," *Big Data Research*, Vol. 2, 2015, pp. 19-27.

Mohammadhossein Barkhordari received the M.S. degrees in Software Engineering from Amirkabir University, Iran. He is currently pursuing the Ph.D. degree in the Information and Communication Technology Research Center, Iran. His research interests include big data, business intelligence, data warehouse, data mining.

Mahdi Niamanesh received his Ph.D. degree in Computer Engineering in the Department of Computer Engineering, Sharif of University Technology in 2009. He is currently a Professor at the Information and Communication Technology Research Center, Iran. His research interests include algorithm pervasive computing, big data.