Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

MSP: Multiple Sub-graph Query Processing using Structure-based Graph Partitioning Strategy and Map-Reduce

Shaik Fathimabi [*], R.B.V. Subramanyam, D.V.L.N. Somayajulu

Department of Computer Science and Engineering, National Institute of Technology Warangal, Telangana, India

ARTICLE INFO

Article history:
Received 14 June 2016
Revised 24 October 2016
Accepted 15 November 2016
Available online 25 November 2016

Keywords:
Graph database
Big data
Structure based graph partitioning
Parallel processing
Map-Reduce
Integrated Graph Index

ABSTRACT

In a distributed environment, the volume of graph database increases quickly because graphs emerge from several autonomous sources. Sub-graph query processing is a challenging problem in distributed environment. Centralized approaches proposed many algorithms, they mine frequent subgraphs from the graph database and construct an index which is very expensive. These algorithms require more number of database scans to mine frequent subgraphs and they use filter and verify approach, which requires many subgraph isomorphism tests. In this paper, we design a novel Map-Reduce based multiple subgraph query processing framework, namely MSP. MSP processes multiple graph queries using distributed index. The framework completely relies on the graph partition and indexing. Moreover, in order to improve its performance, we propose several solutions to balance the workload and reduce the size of Integrated Graph Index. We propose a structure-based partitioning technique and distributed way of building Integrated Graph Index. This work uses two Map-Reduce rounds, the first Map-Reduce round partitions the graphs and creating index for each partition, second Map-Reduce round processes sub-graph queries and index maintenance. A good partitioning will reduce the index size by distributing the load equally to the machines in the cluster and improves the performance of query evaluation. This graph partition and Integrated Graph Index reduces the search space of query graphs. Our approach allows to add data graphs incrementally to Integrated Graph Index while doing query processing. We experimentally show that our approach decreases remarkably the execution time and scales the subgraph query processing to large graph databases.

1. Introduction

Graphs are widely used to model complex structures such as protein interactions, chemical compounds and web data in many applications (Willett, 1998). The contemporary world has huge applications in graph databases. When a database is used to manage the data of objects that are represented by graphs, the database falls into two categories. The first category is a single graph setting where the graph database contains only one large graph. The second category is the transaction graph database that consists of a large number of relatively small graphs. Transaction graph database is used in scientific domains such as chemistry, bioinformatics etc. The graph query processing problems are classified into two types. The first one is the Subgraph query processing which is used to retrieve all the graphs in the database such that a given query graph is a sub-graph of them. The second one is super-graph query processing (Cheng and Ke, 2011) which is used to retrieve all the graphs in the database such that the query graph is a super-graph of them. This paper deals with the subgraph query processing which has a wide range of applications. Sub graph query problem is also known as subgraph isomorphism problem, which belongs to NP- complete (Lubiw, 1981). Many graph datasets are increasing day by day. It is often hard to process large graph database using a single machine because of their size and complexity. PubChem project processes more than 30 million chemical compounds, the storage size of which hits tens of terabytes (Willett, 1998). In recent years the big data phenomenon has emerged in a number of application domains and research including pattern recognition (Kuramochi and Karypis, 2005), social networks, chem-informatics (Willett, 1998), medical image

* Corresponding author.
E-mail addresses: fathimanitw@gmail.com (S. Fathimabi), rbvs66@gmail.com (R.B.V. Subramanyam), soma@nitw.ac.in (D.V.L.N. Somayajulu).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

databases (Euripides, 1997) graph-structured query processing, graph data mining (Xifeng and Jiawei, 2002), close graph (Xifeng and Jiawei, 2003), computational biology.

Meanwhile Map-Reduce (Dean and Ghemawat, 2008) has gained a lot of attention from both industry and academia. MapReduce (hadoop, 0000) provides distributed approach for processing data intensive jobs with no difficulty of managing the jobs across computers. The data centric approach adopted by MapReduce is using the idea of moving computation to data. It uses distributed file system (Shvachko et al., 2010) and (hdfs, 0000) that is particularly optimized to improve the IO performance while handling massive data. Another main reason for this framework is that higher level details are hidden from programmers and allowed to concentrate more on the problem specific computational logic. In a multi user environment, graph queries are given by the number of users. In this paper, we design a novel multiple subgraph query processing, named MSP. MSP solves the problem of processing multiple graph queries over large graph database (Angles, 2012). We propose a distributed graph indexing and query processing method using Hadoop, an open source implementation of MapReduce. We first discuss naive approach which performs all pairwise subgraph isomorphism tests between the graph query set and the graph data set takes long time. To reduce the number of subgraph isomorphism tests, we introduce Structure-Based Partitioning and Integrated Graph Index, which is used to do both filter and verify the steps. Index maintenance is easier by using the Integrated Graph Index. The contributions of our work are summarized below:

- We propose an efficient multiple subgraph query processing framework, MSP which can handle large- scale graph database and set of query graphs.
- We propose a simple but effective workload-aware distribution strategy, Structure-Based data partition technique using MapReduce to enhance the default data partition technique provided by MapReduce.
- We introduce an efficient approach to build distributed way of constructing Integrated Graph Index uses multiple edges and vertex label based join.
- We design distributed algorithm to process queries and index maintenance in one Map-Reduce round.
- We introduce maximum depth first code for Structure-Based partition.
- We experimentally demonstrate the performance of the algorithm on synthetic as well as real world large data sets.

The rest of the paper is organized as follows. We discuss related work in Section 2. Section 3 presents preliminaries. The proposed methods are presented in Section 4. Experimental results are discussed in Section 5. Finally, we conclude the paper in Section 6 (Table 1)

2. Related work

The use of graphs has become increasingly significant in modeling complicated structures. Now-a-days graphs are used to model any kind of data. There exist many algorithms for solving the centralized and in-memory version of sub- graph query processing task, most notable among them are Graph Grep (Giugno and Shasha, 2002), Tree and graph search (Shasha et al., 2002), FGIndex (Cheng et al., 2007), GIndex (Yan et al., 2005), ClosureTree (He and Singh, 2006), TreePi (Zhang et al., 2007), graph indexing interms of tree + delta (Zhao et al., 2007), a novel spectral coding in a large graph database (Zou et al., 2008), GString (Jiang et al., 2007) Graph containment and indexing is proposed in Chen et al. (2007). These algorithms assume that the dataset is small and the mining task finishes in a stipulated amount of time using an in-memory method.

Table 1
Notations.

Symbol	Description
| g |	Size of the graph, number of edges in the graph
D	Graph database
G	Set of graphs in graph database or particular partition
Q	Set of query graphs
Aqi	Answer set of a query qi
IGI	Integrated graph index of graph database
IGIpno	An integrated graph of particular partition of D
host(e)	Set of graphs (IDs) that currently share e in *IGIpno*
freq(e)	Number of graphs that have e in *IGIpno*

In (Kim et al., 2013) the Parallel Processing of Multiple Graph queries using Map-Reduce (PPMG) approach extracting features from data graphs every time to process query graphs. It takes a long time. Luo et al. (2011) solved a single subgraph query processing problem using an edge index.

Now-a-days many algorithms are processing a single large graph using Map-Reduce. Pregel (Malewicz et al., 2010) proposed vertex centric graph processing. Kang et al. (2011) developed a system to analyze a single large graph such as web data or social network. In (Kang et al., 2008) diameter estimation and mining with Hadoop is described. In this paper, we focused on a large set of small graphs. Solving the task of sub graph query processing on distributed platforms like Map-Reduce is challenging for various reasons. Two types of Data Partitions are available in the literature

- Default Graph Partitioning
- Density-Based Graph Partitioning

2.1. Default graph partitioning

It is the default split method used by Map-Reduce. It partitions the graphs based on chunk size and it does not consider the characteristics of the input data graphs during partitioning. The number of graphs is unequally distributed and load is not distributed uniformly. However, for datasets where the size of the graphs in a dataset varies substantially, we use density-based graph partition.

2.2. Density-Based graph partitioning

Density-Based Graph Data Partitioning is presented in Aridhi et al. (2015). This partition is suitable for datasets, where the size of the graphs varies. It is based on characteristics of the input data graphs, i.e. density of each graph during the creation of partitions. It distributes the load equally to all machines on the cluster based on density of graphs and not based on the labels on the edges and structure of the graphs.

3. Problem definition

In this work we consider undirected, labeled and connected graphs. We formally define a graph and the subgraph query problem which is solved in this paper. Then we describe the representation of graph data in Map-Reduce.

3.1. Definitions

Graph: A graph is denoted by a tuple g = (V, E, L, l) where V is the set of vertices and E is the set of undirected edges such that

$E \subseteq VxV$. L is the set of labels of vertices or edges, and the labeling function l defines the mapping: $V_U\ E \rightarrow L$. We also denote the vertex set and the edge set of graph g by V(g) and E(g) respectively. We define the size of a graph g, denoted as |g|, as the number of edges in g, that is | g | = | E(g) |.

Subgraph Isomorphism Given two graphs $s = (V_s, E_s, L_s, l_s)$ *and* $g = (V_g, E_g, L_g, l_g)$, s is said to be sub-graph isomorphic to g ($s \subseteq g$) if and only if there exists an injective function $f: Vs \rightarrow Vg$ such that (1) $\forall v \in Vs$, we can have $f(v) \in Vg$ and $ls(v) = lg(f(v))$; (2)$\forall(u, v) \in Es$, we can have(f (u), f (v)) $\in$ Eg, and fs [u, v] = fg [f (u), f (v)].

3.2. Problem statement

Let $D = g_1, g_2, g_3 \ldots g_n$ be a graph data set. Furthermore, let $Q = q_1, q_2, q_3 \ldots q_x$ be a graph query set such that $|Q| \ll |D|$ for each graph query q $\in$ Q, we find all the graphs to which q is subgraph isomorphic from D. The result is $A_Q = A_{q1}, A_{q2}, \ldots A_{qx}$ where $A_{qi} = g_j{:}g_j \in D$, $q_i \subseteq g_j$ i.e. each A_{qi} contains the set of data graphs in D that are super- graphs of q_i .

This problem varies from existing sub-graph query processing problem which processes one subgraph query at a time, but here it processes a batch of queries at a time. The reasons why we develop a solution to process batch of queries at a time are as follows:

In a distributed environment, more queries are coming from different sources at a time. Processing of queries that come in as a high speed stream is useful for many applications that require prompt query response. Here we integrate subgraph query processing and index maintenance.

Moreover, batch query processing enables us to eliminate the repeated process of common parts among queries and insert graphs, so as to obtain a higher throughput.

Our goal in this paper is to develop an efficient distributed system for processing multiple sub-graph queries using Integrated Graph Index.

3.3. Graph representation

Graph as a Single Line: Most of the Graph datasets are available in the multi-line format. Some existing frequent subgraph mining works (Mansurul and Mohammad, 2013) using Hadoop divides the graphs into a set of files in the data preparation step and the set of files are given as input to the MapReduce program. In this work, we convert multi-line format of a graph into a single-line format of graph. In a single-line format there exists a serialized format g, which enumerates vertices and edges in g, i.e {| $V(g)$ |, | $E(g)$ |, $l(V(g))$, $E(g)$} where e $\in$ E(g) is represented as from-gid, to-gid, l(e). <graphid>,<no_of_vertices>,<no_of_edges><Labels of all vertices>,<edgelist>

Graph id: a unique identifier for a single graph g.
No of vertices: total number of vertices in the graph.
No of edges: total number of edges in the graph.
Labels of all vertices: List of Labels of all the vertices of the graph.

Edgelist: List of edges of the graph. Each edge contains three elements.<sourceid,destinationid,edgelabel> For example: g2,4,4, A,B,C,E,0,1,b,0,2,d,1,2,e,2,3,f. [Grouping as 3 pair will give a nice example].

4. Proposed approach

In this section, we discuss the proposed approach for multiple sub-graph query processing using structure based partitioning and Integrated Graph Index in a distributed environment. The large graph database has different types of graphs having different labels. The first step is partitioning the graphs based on labels and structure. The second step is construction of an Integrated Graph Index for the partition of graphs. The third step is multiple graph query processing using Integrated Graph Index.

First, we discuss the following Hadoop methods to design an efficient algorithm using Map-Reduce:

In Mapper Combiner Design Pattern (IMCDP): In this pattern Combiner is written inside the mapper logic. Instead of sending a line by line output, the mapper does local aggregation by using in mapper combine. It does not change the running time complexity of the algorithm, but greatly reduces both the number and size of key-value pairs that need to be shuffled from the mappers to the reducers. Because of this reason, we use IMCDP.

Distributed Cache: It distributes application-specific large, read-only files efficiently to all mappers. It is a facility provided by the MapReduce framework to cache files required by all mappers, i.e. to all task trackers (nodes). Using distributed cache we can distribute some common data across all task trackers. When we need to distribute a file, multiple copies of the file would be maintained for all task trackers to access.

In this paper the entire work is divided into four phases

- Graph partition
- Integrated Graph Index Creation
- Processing of set of sub-graph queries and Index Maintenance

The above steps are discussed in the following subsections.

4.1. Graph partitioning

4.1.1. Motivation and principle

The motivation behind dividing the input large graph database into partitions is to get all similar graphs into one partition. In a distributed environment, the efficiency of the algorithm depends on how efficiently, we divide the data graphs and distribute them to nodes on the cluster. A good partition technique distributes the data efficiently and reduces the integration cost and the computation cost i.e. subgraph isomorphism testing cost. Some authors divide the data as a pre-processing step on a single machine. But in this paper, we divide the data graphs using Map-Reduce phases. To process the graph queries, we need to check all the data graphs which are available on the cluster. Instead of checking all the graphs on all machines, we can check the graphs having similar structure of query graphs. This approach is more efficient in terms of both communication cost and computation cost. Graph partition techniques are useful to get all the similar graphs to one machine. With this motivation we develop a structure based graph partition approach. In a distributed environment, heterogeneous data come from different sources to process. If we know the Domain knowledge well in advance, we can partition the graphs based on domain knowledge. Domain based partition is more efficient for indexing and query processing.

We propose two Graph Database partition techniques based on labels and the structure of the data graphs as follows:

- Label-Based Graph Partition
- Structure-Based Graph Partition

4.1.2. Label-Based graph partition

In this partition the graphs having the same labels get into one or more partitions. One Map-Reduce round is required for this partition. Mapper reads a graph and prepares key and value, and the key is the concatenation of all edges according to the lexicographical order and value is the concatenation of graph id and depth first traversal code of the graph. Each edge is represented by three elements:

<source vertex label, edge label, destination vertex label>.

For lexicographic order, we first consider the source vertex label, then edge label and at the last destination vertex label. The reducer receives all graphs having similar types of label builds Integrated Graph Index and stores in HDFS. Algorithm 1 shows the procedure of label based graph partition. Fig. 1a shows the graphs after label partition.

4.1.3. Structure based partition

In some applications, labels of vertices and edges are the same, but the structure of the graphs is different. We propose a structure based graph partition technique to bring all the graphs which have similar labels and structures to one machine. We use both label and the structure of the graph as a feature to partition the graphs. To get the structure we use maximum depth first traversal code of the graph. Based on maximum depth first traversal code, we partition the graphs. So it is more efficient compared to label based partition. Two Map-Reduce jobs are used to do structure based graph partition. The first Map-Reduce round is used to find each edge and its frequency. Second Map-Reduce round is used to partition the graphs based on maximum dfs code and to do graph integration. Fig. 4 shows the overview of the structure based partition. Fig. 1a shows the graphs after structure based partition. In this partition we use distributed cache to store the result of first Map-Reduce round and to read in second Map-Reduce round. Second Map-Reduce round does structure based partition.

First Map-Reduce round is used to find the frequency of each edge: In Algorithm 2 the Mapper uses IMCDP approach to do the local aggregation at the mapper. Mapper reads the graph and parses the edges and does local frequency count of edges for the entire partition. It sends the edge value as key and local frequency count as value. The reducer does the global aggregation of frequency of edge and sends the edge as key and frequency as value.

Maximum Depth First Traversal code: Depth First Traversal code of the graph starting from the edge having highest frequency then select next highest frequency edge. Backward edges are considered first, and then forward edges. All rules are the same as the depth first search code in gspan (Xifeng and Jiawei, 2002). If two edges have the same frequency then select the edge according to the lexicographic order of edge label.

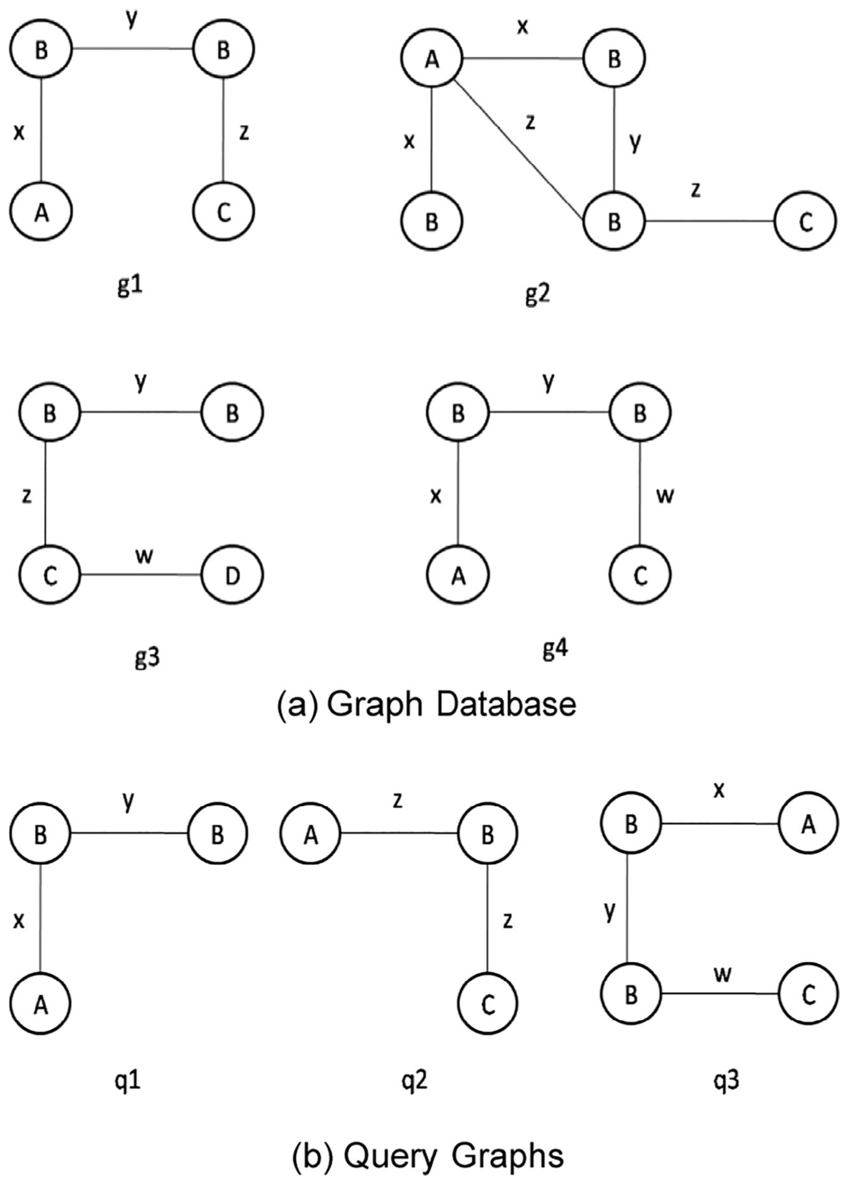


Figure 1. An example graph database and query graphs.

4.2. Efficient way of creating Integrated Graph Index

We used the approach proposed in Cheng and Ke (2011) to integrate the graphs as an Integrated Graph Index. In addition to their algorithm, we introduce two optimizations during the integration to reduce the size of Integrated Graph Index. Given a set of graphs G, the concept of graph integration is to merge all the graphs in G into a single compact graph IGI, where the repeated common substructures of the graphs are eliminated in G as much as possible.

The two optimizations proposed to reduce the size of Integrated Graph Index are

- Parallel Edges
- Vertex Label based Join

Algorithm 1: Label-Based Graph Partition and Integrated Graph Index construction

```
Result: Integrated Graph Index files
Data: Large graph database D
Class Mapper
  method setup()
  initialize integrated graph IG
  method map(N:Offset , V:[gid,gcode])
  begin
      parse V add vertices into g add edges into g
      for each edge labeled with EdgeLabeli in V do
          vertexlabels=concatenate unique vertex
          label
          edgelabels=concatenate unique edge label;
          key=vertexlabels+edgelabels;
          dfscode=g.dfs()
      end
      value=gid+dfscode;
      emit(key,value)
  end
Class Reducer
method reduce(Key: feature ,V: Set of dfscodes)
begin
    sum=0
    for each dfs in V do
        integrategraph(dfs)
    end
    method close()
    write integrategraph to HDFS
end
```

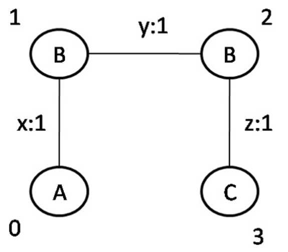
4.2.1. Parallel edges

Between two vertices two or more edges exist which are called Parallel Edges. We can represent parallel edges as a single edge with multiple edge labels. While integration of Vertex labels is matched, edge labels are different therefore, instead of adding separate vertex and edge in the integrated graph, we can add parallel edges to the vertices. This will reduce the Integrated Graph Index size. The labels and pointers are stored in header table.

4.2.2. Vertex label based join

Depth First Traversal is not matched during integration use vertex label to integrate the graphs this is called Vertex Label based join. In this approach if the vertex label and edge label are not same in the integration process, instead of adding new vertex, it connects to the vertex with this label in the existing integrated graph. It reduces the number of vertices in the graph by utilizing the existing vertices. Fig. 5 shows the example to perform vertex label based join. The methods required to initialize integrated graph and Integrated Graph Index creation are shown in Algorithm 4. For the example graph database given in Fig. 1a and set of query graphs given in Fig. 1b, the integration process is explained. Fig. 2 is the IG after g1 is placed into IG. Fig. 3 shows the IGs after integration of g2, g3, g4 and we can identify the multiple edges. Fig. 5 shows an Example for Label based join.

Normal approach of graph integration is to first mine frequent subgraphs from G and then merge the graphs in G by sharing their frequent subgraphs in descending order of frequency. However frequent subgraph mining is costly, especially when database update is frequent or queries come as a stream.



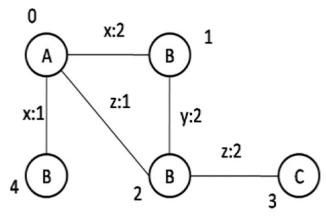
Edge Table

Edge	Freq	Pointers
A B x	1	(0, 1)
B B y	1	(1, 2)
B C z	1	(2, 3)

Figure 2. IGI1 with g1.

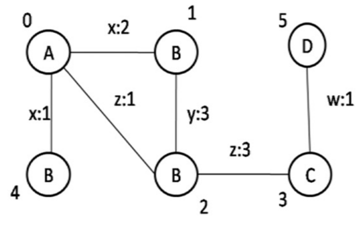
Algorithm 2: Algorithm to find Frequency of each edge

```
Result: Edge and its frequency
Data: Large graph database D
Class Mapper
AssociativeArray edgemap;
method map(N:Offset , V:[gid,gcode])
begin
    StringTokenizer itr = new
     StringTokenizer(value.toString());
    for each edge labeled with EdgeLabeli in V
    do
    if ( edgemap.containsKey(EdgeLabeli ) ) then
        int total = edgemap.get(token).get() + 1;
        edgemap.put(token, total);
    end
    else
        edgemap.put(token, 1);
    end
    cleanup()
    begin
        for each edge in edgemap do
            sKey = edge.getKey();
            total = edge.getValue();
        end
        context.write(sKey, total);
    end
end
Class Reducer
method reduce(EdgeLabel e,V: List of totals)
begin
    sum=0
    for each i in V do
        sum=sum+i
    end
    emit (EdgeLabeli,sum)
end
```



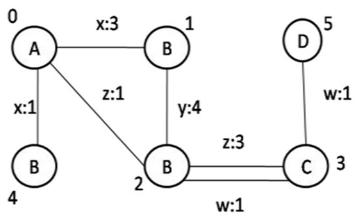
Edge Table		
Edge	Support	Pointers
A B x	2	(0, 1) , (0, 4)
B B y	2	(1, 2)
B C z	2	(2, 3)
A B z	1	(0,2)

(a) IGI$_1$ after merging g$_2$



Edge Table		
Edge	Support	Pointers
A B x	2	(0,1)(0,4)
B B y	3	(1,2)
B C z	3	(2,3)
A B z	1	(0,2)
C D w	1	(3,5)

(b) IGI$_1$ after merging g$_3$



Edge Table		
Edge	Support	Pointers
A B x	3	(0,1)(0,4)
B B y	4	(1,2)
B C z	3	(2,3)
A B z	1	(0,2)
C D w	1	(3,5)
B C w	1	(2,3)

(c) IGI$_1$ after merging g$_4$

Edge with Id	Graphs ids
0 1 A B x	G1,G2,G4
1 2 B B y	G1,G2,G3,G4
2 3 B C z	G1,G2,G3
0 2 A B z	G2
0 4 A B x	G2
2 3 B C w	G4
3 5 C D w	G3

(d) Index Table

Figure 3. IGI1 and its edge table and index table.

We propose an efficient algorithm to merge a set of graphs into a compact graph by utilizing the statistics of the edge frequency of the graphs in G. Let G be a set of graphs and IGI be the compact graph of G, called as Integrated Graph Index (IGI). We keep the information of the graphs of G at the edges in IGI, while eliminating duplicate edges shared among edges of G. We first define the frequency of an edge e in IGI denoted as freq(e), as the number of graphs in G that share e in IGI. For the purpose of query processing, we also associate with each edge in IGI the set of graphs (IDs) in G that share e, denoted as host(e). The basic idea of graph integration is to use the frequency of the edges in the current IGI to guide the merging of an incoming graph into IGI. More specifically, when merging a graph g into IGI, we find all the edges in g that are also in IGI and pick the one edge that has the highest frequency in IGI, we simply break the tie by the lexicographic order of the edge labels. Then using this edge as starting edge in both g and IGI, we perform a simultaneous depth-first traversal of both g and IGI to find their common subgraph.

Let e_0 be the starting edge and e_1 be the next edge to visit in the depth-first traversal of g. Let E_1 be the set of edges that we can choose to visit next to e_0 in the depth-first traversal of IGI. We find an edge in E_1 that matches e1 to visit. If there are multiple edges in E1 matching e1, we choose the one with the highest frequency to visit. This process continues until we meet an edge in g that cannot be matched in IGI. The matched edges in the simultaneous depth-first traversal form a common subgraph of g and IGI. We merge g into IGI by sharing their common subgraph, while we create new edges in IGI for those edges in g that have not been matched. The matched edges in the simultaneous depth-first traversal form a common subgraph of g and IGI. We merge g into IGI by sharing this common subgraph, while we create new edges in IGI for those edges in g that have not been matched.

Note that we match edges by (lu, le, lv) i.e the definition of a district edge. There may be multiple instances of the distinct edge e_0 in g. In this case, we run the simultaneous depth-first traversal multiple times starting at each instance of e_0 in g. Among the multiple traversals, we pick up the largest common subgraph of g and IGI, and we merge g into IGI by sharing this subgraph. During this process we use a common subgraph with parallel edges and vertex label based join. The parallel edges are useful to reduce the number of edges of IGI and vertex label based join is used to reduce the number of vertices in IGI.

To find the edge that has the highest frequency in IGI as a starting edge for the simultaneous depth-first traversal, we construct an edge table to keep the set of distinct edges in IGI. Each distinct edge e_d in the edge table has a list of pointers to the instances of e_d in IGI and the first pointer is to the instance which has highest frequency. Algorithm 4 presents the construction of Integrated Graph Index (IGI). For each incoming graph g_i, the algorithm first finds the frequency of each distinct edge of g_i from the edge table and then picks up the edge e_0 that has the highest frequency, where e_0 points to its instance e in IGI. Then for each instance e_1 of e_0 in g_i, the algorithm finds the largest common subgraph of g_i and IGI. For each subgraph we apply vertex label based join and multiple edges techniques. We then pick the largest matching subgraph g and merge g_i into IGI by sharing g. Then, a corresponding new edges is created in IGI for each edge in g_i but not in g. During the merge, for each edge in g_i, we also increment the frequency and update the index of its matching edge in IGI to assist future integration. It may be noted that the graph IDs in each index(e) is automatically sorted since the graphs are merged into IGI in the ascending order of their IDs. Finally IGI is outputted when all the graphs in IGI are merged.

The merge of each g_i into IGI takes only linear time in the size of gi, assuming the number of instances of a distinct edge in gi is a constant for most datasets. The total complexity of Algorithm 4 is $O(n|G|)$ where n is the average size of the graphs in G. In the worst case, when every graph in G consists of only one distinct edge (ie all edges are identical), the complexity is $O(n^2|G|)$. However, even n^2 is small for graphs in a transaction graph database.

Example 1: Fig. 1 shows a set of graphs G that consists of four graphs g_1,g_2, g_3 and g_4. Initially, the integrated graph IGI = g_1, which is shown in Fig 2. IGI x:1 means that the edge has a label x and

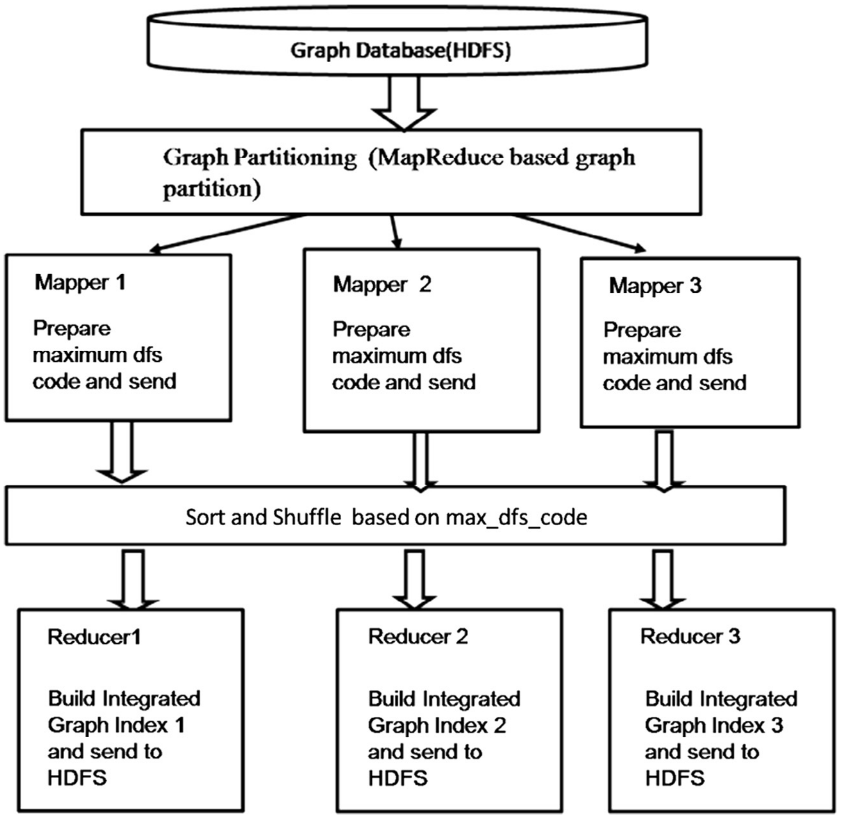

Graph Database(HDFS)
Graph Partitioning (MapReduce based graph partition)
Mapper 1 Prepare maximum dfs code and send
Mapper 2 Prepare maximum dfs code and send
Mapper 3 Prepare maximum dfs code and send
Sort and Shuffle based on max_dfs_code
Reducer1 Build Integrated Graph Index 1 and send to HDFS
Reducer 2 Build Integrated Graph Index 2 and send to HDFS
Reducer 3 Build Integrated Graph Index 3 and send to HDFS

Figure 4. An overview of structure based graph partition and Creation of Integrated Graph Index: The result of algorithm 2 is loaded into distributed cache for structure based partition. Algorithm 3 shows the procedure to do structure based partition and creation of Integrated Graph Index. In Algorithm 3, the Mapper reads the graph and prepares maximum DFS code to send Maximum DFS code as key and the value is concatenation of graph id and Maximum DFS code.

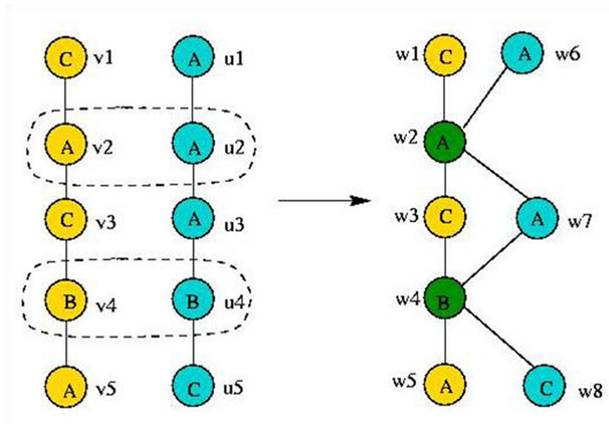


Figure 5. Example of Label based join.

frequency 1. Fig. 3 shows the IGI after integration of g2, g3 and g4 in Fig. 3a–c respectively, and edge table with pointers. The index table for IGI is shown in Fig. 3d.

The graph integration method is simple and efficient. It has several advantages.

- By allowing the descending order of edge frequency when merging the graphs, we are able to integrate the graphs into the position that many other graphs are integrated into. This approach uses the principle of using frequent subgraphs as the integration guidance to extract the common sub-graphs of many graphs.
- Graph integration approach is very fast since it does not involve any expensive operation such as frequent subgraph mining or subgraph isomorphism test. The most costly step is to perform the depth-first traversal that is linear in size of the graph g. The edge frequency used to guide the integration can be easily collected and maintained during the integration process.
- Integrated Graph Index keeps all neighborhood information of the graphs. By utilizing Integrated Graph Index we can extract common subgraphs.

4.3. Graph query processing and index maintenance

We now discuss how to do query processing and index maintenance using IGI. This step does both the query processing and index maintenance simultaneously. Instead of doing query processing and index maintenance separately, here we combine both steps in one step to eliminate the redundant work and number of Map-Reduce rounds. We first give the overall framework and then present the details of each step. In this phase the required Integrated Graph Index files are loaded from HDFS. And another input is query graphs and new graphs to insert. Algorithm 5 shows the entire process required for this phase. The framework of our query processing system consists of three major steps as follows:

Algorithm 3: Structure-Based Graph Partition and Integrated Graph Index Creation

```
   Result: Structure-Based Graph Partition
   Data: Data: Large graph database D
1  Class Mapper
2  begin
3      method setup()
4      begin
5          load edge and frequency into distributed cache
6          initializeIGI()
7      end
8      method map(N:Offset , V:[gid,gcode])
9      begin
10         parse V and place into graph g
11         maxdfscode=g.getmaxdfscode()
12         key=hash(maxdfscode)
13         emit(key,g.graphid+maxdfscode)
14     end
15 end
16 Class Reducer
17 begin
18     method reduce(Key: feature ,V: List of
        maxdfscode)
19     begin
20         sum=0
21         for each mdfs in V do
22         integrategraph(mdfs);
23     end
24     method close()
25     begin
26         write headertable to HDFS
27         write hosttable to HDFS
28     end
29 end
```

- Partition of input graphs based on Integrated Graph Index feature.
- Input graphs Integration.
- Query graph verification and index maintenance.

4.3.1. Partition of input graphs based on Integrated Graph Index feature

When the sequence of query graphs and new graphs to insert are issued, we first need to do preprocessing step. Here input graphs mean both query graphs and new graphs to insert. To process the input graphs first partition the input graphs according to the structure of IGIs.

Feature Table: Depth First Traversal of all IGIs is placed in a table and stored in a file. The format of Feature table is feature and its file name, which contains that particular IGI. Table 2 shows the format of Feature table.

Filter the IGIs: The number of input graphs is partitioned based on the structure of IGIs which is in Feature Table. From the feature table we get maximum dfs code of each IGI. For each query graph we check if all edges of query exist in that IGI or not. If all edges do not exist in any IGI then we can filter that query. That means the result for that query graph is null. For new graphs to insert into IGI we check all the edges of new graph in IGI we select which IGI has more edges of that new graph. Based on edges existence, the graphs are divided and placed in a table. The result of input graph partition is stored in graph Partition table. The format of graph partition table is shown in Table 3.

Graph Partition Table: After filter step the result is stored in Graph partition table and its format is shown in Table 3.

Table 2
Feature table for IGIs.

Feature	IGI*pno* File Name
1 2 B B y 2 3 B C z 3 5 C D w 3 2 C B w 2 0 B A z 0 1 A B x 0 4 A B x	IGI1 filename
Maximum dfs code of IGI2	IG2 filename

Table 3
Graph Partition Table.

IGI file name	Graph Ids
IGI1 filename	Q1, Q2, Q3, G150
IGI3 file name	Q4, Q5, G100

Table 4
Result of query processing.

Query Id	Graph Ids
Q1	G1, G2, G4
Q2	G2
Q3	G2

4.3.2. Input graphs integration

After the input graphs partition apply Algorithm 4 to integrate all the input graphs that belong to each IGI. For each group of queries we get one Query Integration Index. Query integrated index graphs are loaded into distributed cache for the next step verification.

4.3.3. Verification

During verification step we load only required IGIs based on preprocessing step. We load only these Integrated Graph Index files from HDFS instead of all IGIs. Load the required IGIs from HDFS. The integrated Graph Index files are loaded into a number of machines. All the input graphs are loaded into distribute cache to make them available to all machines and the result of preprocessing is loaded into distributed cache i.e shown in Table 3. According to the IGI filename get all the query numbers. During query processing verify if the structure of query exists in IGI or not. If it exists then send it to the answer. In case of new graphs insert graph ids in edge index table and edge table. When we add any new graphs to IGI at the end we will store updated IGI on HDFS. For the example given in Fig. 1, the final result is shown in Table 4. Instead of getting edges for each query from IGI, when we process the first query, the edges <*ABx*>, <*BBy*> and their graph ids are placed in table and are used for further queries and this will reduce the I/O time.

5. Experiments

This section presents an experimental results that demonstrate the performance of our approach on synthetic and real datasets. In the following experiments, we aim to determine the efficiency of Integrated Graph Index on centralized approach and distributed approach. In particular, we compare index creation time, query processing time and index maintenance cost. It first describes the used datasets and implementation details. Then, it presents a discussion of the obtained results.

Algorithm 4: Methods to create Integrated Graph Index

```
Result: Integrated Graph Index and edge table and
        index table
Data: Data: Set of graphs in the form of dfs code
1  method initializeIGI()
2  begin
3      intialize integrated Graph as IGI
4      initialize Edge Table and Index Table
5  end
6  method integrategraph(dfscode G)
7  begin
8      if (IGI.numberofgraphs=0)
9      add G to IGI
10     else
11     Find from the edge table the distinct edge in G
        that has the highest frequency in IG
12     Let e0 be this edge and it points to e in IGI
13     for each instance e1 of e0 in G do
14     Match G with IG by a depth first traversal
15     Starting from e1 in G and e in IG
16     For each edge e11 in G in depth-first traversal do
17     If(more than one edge in IG match e11 )
18     Choose the edge with the highest frequency
19     If any mismatch in edge label but vertex labels are
       matched
20     then add multiple edges
21     If edge label and vertex label are not matched then
       add edge to that vertex as label based join
22     Let sub be the largest matching subgraph of G and
        IG obtained in Lines 10-17
23     Merge G into IG by sharing sub
24     for each edge instance e of G i merged into IG do
       do
25     Increment freq(e)
26     Add graphid of G to index(e);
27 end
```

5.1. Experimental setup

5.1.1. Datasets

The datasets used in our experimental study are described in Table 5 Real world graph datasets which are taken from an online source that contains graphs extracted from the PubChem website. PubChem contains one million chemical structures. Each graph has 23.98 vertices, 25.76 edges, 3.5 distinct vertex labels, 2.0 distinct edge labels on average, and the total number of distinct vertex labels and distinct edge labels is 81 and 3, respectively. The size of PubChem dataset is 434 MB. For our tests, we generate 3 lakhs graphs. We also randomly generate several sets of graph queries and new graph to insert, which contain various numbers of queries ie | Q | = 10, 100, 200, 300 up to 2000.

Table 5
Real life biological datasets.

Graph dataset	Number of graphs	Average size of each graph
Yeast	79,590	23.2
P388	41,470	26
SN12C	40,002	31.2
OVCAR-8	40,514	31.3
NCI-H23	40,351	31.5
MOLT-4	39,763	30
PC-3	27,507	32
SF-295	40,269	32
SW-620	40,530	31

Algorithm 5: Query Processing and Index maintenance using Integrated Graph Index

```
Result: Each query and List of graph ids
Data: set of query/ insert graphs and result of
       preprocessing step , Integrated Graph Index files
1  Class Mapper
2  method setup() begin
3      load result of preprocessing step from distributed
       cache load query graphs into rtable load insert
       graphs into instable
4  end
5  method map(N:Offset , V:[gid,gcode])
6  begin
7      read integrated graph into memory and place in IG
8      querylist= get querylist from rtable for this
       IGfilename
9      for each query in querylist begin
10         if (all edges exists )
11         then if( verify query IG)
12         local result=intersect all graphids
13         emit (queryid,local result)
14     end
15     for each insert graph in instable begin
16         Update Integrated graph index by adding
           insert graphs
17     end
18 end
19 method close() begin
20     store updated IG to HDFS
21 end
22 class Reducer
23 method reduce(key: query id, value : list of graph ids)
24 begin
25     for each list in value do
26         globallist=globallist+list;
27     end
28     emit (queryid, graphidslist)
29 end
```

5.1.2. Implementation platform

We implement this work in Java and using Hadoop (version 1.2.1) an open source version of Map-Reduce. The database files are stored in the Hadoop Distributed File System(HDFS) an open source implementation of GFS (Ghemawat et al., 2003). All the experiments of our approach were carried out using a local cluster with 9 nodes. The processing nodes used in our tests are equipped with a Hardware:1 + 8 Node Cluster **Front-end:** HP Proliant DL380P Gen8, 2 x Intel xeon CPU E5-2640 (2.5 GHz/ 6-core/15 MB/95w) processor, 64 GB RAM, 333 X 600 GB HDD machine; Storage: HP MSA2040 SAN SFF/ 24 x 300 GB HDD/ 8∗16 GB POETS; OS: Rocks Cluster 6.1.1 + CentOS 6.5 Server with Hadoop1.2.1.

Data Node: Intel xeon E5-2640 (2.5 GHz / 6-core/15 MB/95w) processor, 16 GB RAM, 2 X 300 GB HDD machines.

5.2. Experimental results

5.2.1. IGI creation on centralized vs distributed platform

This experiment shows the time variation for centralized approach and distributed approach. Centralized approach means only single machine without using Hadoop. Single machine configuration:Intel i3 processor, 8 GB RAM, 500 GB Hard disk. For this experiment we used synthetic data set consisting of 12,000 graphs. Table 6 shows the time taken for centralized and distributed approach. This shows distributed approach is scalable to large graph datasets.

5.2.2. Query processing time on centralized vs distributed platform

In this experiment we conducted query processing on 12,000 data graphs which are shown in Table 5 with different numbers of query graphs. It takes less time for hadoop and it scales well to large graph databases. Table 7 shows the time versus number of query graphs. As the number of queries increase the time also increases.

5.2.3. Integrated Graph Index building time versus number of data graphs

In this experiment we used synthetic dataset consisting of 12,000 graphs and we executed on 9 node cluster machine. We first test the Integrated Graph Index creation time for number of data graphs. Here the time is initially not increasing because even if we increase graphs the data are distributed to a number of machines so it takes the same time. Fig. 6 shows the time variation according to the number of data graphs.

5.2.4. Integrated Graph Index creation time using label based partition vs structure based partition

In this experiment we used synthetic dataset consists of 12,000 graphs and we executed on 9 node cluster machine. This experiment is conducted using both Label based partitioning algorithm and Structure based partition algorithm. In this experiment we used synthetic datasets shown in Table 5 and we executed on 9 node cluster machine. Fig. 7 shows the time taken for LP and SP techniques. LP is taking less time compared to SP partition.

5.2.5. Query processing time for label based partition versus structure based partition

This experiment is conducted to compare Label Based partition and structure based partition. Fig. 8 shows the comparison of query processing when we use label based partition and structure based partition. Structure based partition takes less time compared to Label based partition.

Table 6
Integrated Graph Index creation on centralized vs distributed.

Dataset Size (Number of graphs in thousands)	Time taken for Centralized Algorithm in minutes	Time taken for Hadoop (minutes)
0.5	0.6	0.66
1	2	0.7
2	4	0.8
4	7	1.5
6	9	3.42
8	15	5.2
12	25	7.3

Table 7
Query processing time on centralized platform vs distributed platform.

Number of query graphs	Time taken for Centralized Algorithm in mins	Time taken for Hadoop
1	0.05	0.04
10	18	14.6
40	47	19
80	85	26
100	105	30

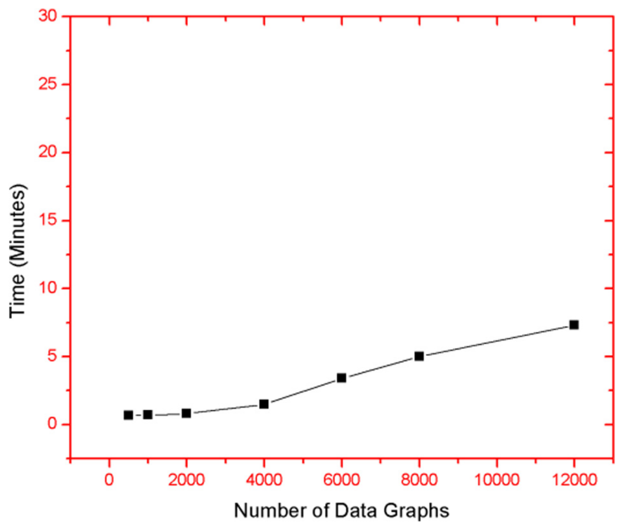


Figure 6. Integrated Graph Index building time versus number of data graphs.

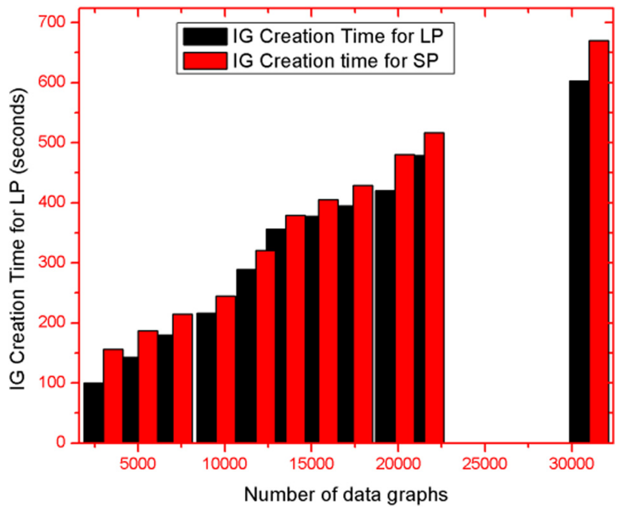


Figure 7. IGI creation time using label based partition vs structure based partition.

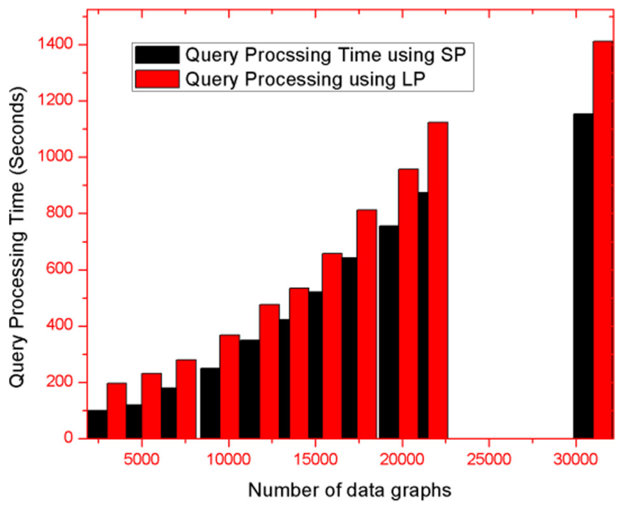


Figure 8. Query processing time using label based partition vs structure based partition.

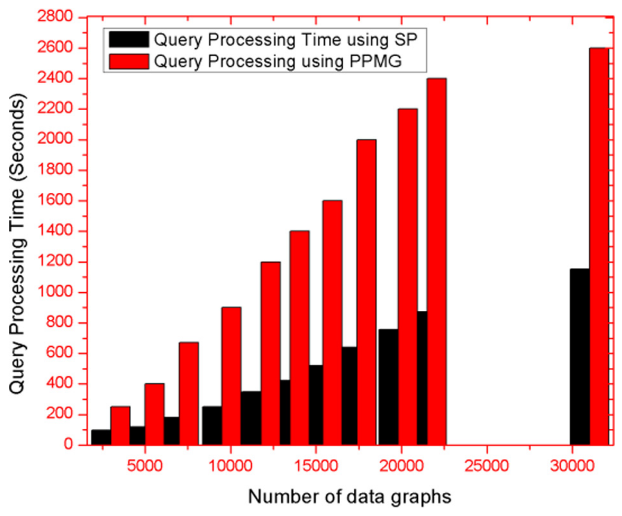


Figure 9. Query processing time using structure based partition versus PPMG.

5.2.6. *Query processing time for structure based partition versus existing algorithm PPMG*

In this experiment we used synthetic datasets and we executed on 9 node cluster machine. This experiment is conducted to compare Structure Based Partition and PPMG (Kim et al., 2013). Fig. 9 shows the comparison of query structure based partition and PPMG. There is lot of difference in the time taken for PPMG and our approach. Our structure based partition takes very little less time compared to earlier algorithm PPMG.

5.2.7 Integrated Graph Index creation using REAL datasets

In this experiment we used different real datasets and recorded running time for Integrated Graph Index creation. We consider five real datasets and showed the time required to create the Integrated Graph Index. The time is shown in Fig. 10.

5.2.8. *Query processing time analysis using real datasets*

In this experiment we used real datasets shown in Table 5 and we executed on cluster machine. Here we used different numbers

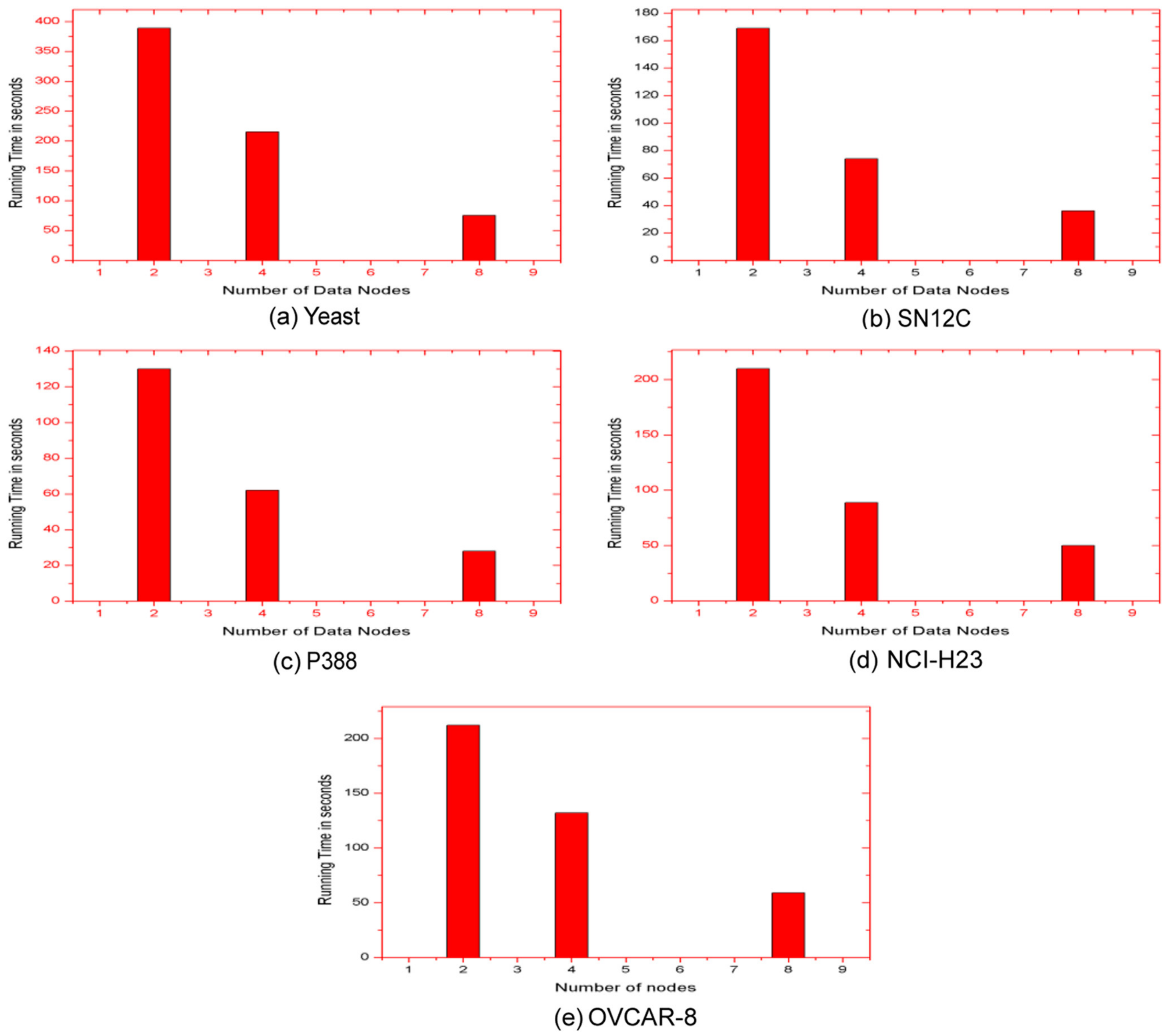


Figure 10. Performance evaluation on Integrated Graph Index creation versus number of data nodes.

of query graphs and we recorded execution times. It is shown in Fig. 11. We executed for the number of query graphs. If query graphs are more then running time is more.

5.2.9. Integrated Graph Index creation time versus graph partitioning technique

In this experiment we analyzed different graph partitioning techniques and running time to create Integrated Graph Index. We used real datasets shown in Table 5. We executed on 9 node cluster machine. Here we used four partitioning techniques default graph partition, density based graph partition, label based graph partition and structure based graph partition. In this experiment we recorded Integrated Graph Index creation time for different partition techniques. Here we compared four partition techniques. Default partition is taking long time compared to all other partition techniques. Structure based partition is taking least time compared with label based partition and density based partition, it is shown in Fig. 12.

5.2.10. Query processing versus graph partitioning technique using real datasets

In this experiment we analyzed the query processing time for different partition techniques. Here we used real datasets shown in Table 5. We executed on 9 node cluster. We recorded the running time for query processing here we used 100 queries. For structure based partition is taking less time compared to all other partition techniques. It is shown in Fig. 13.

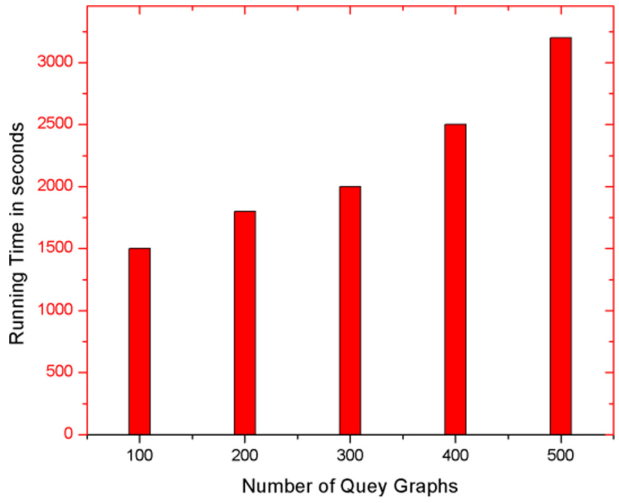


Figure 11. Number of query graphs versus running time.

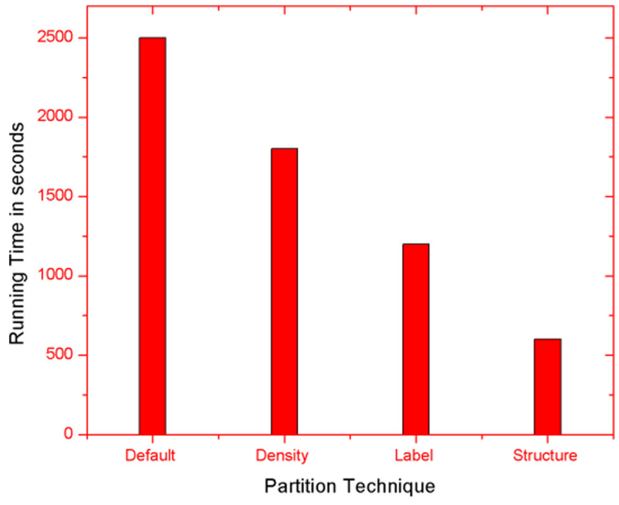


Figure 12. Integrated Graph Index creation time versus graph partition technique.

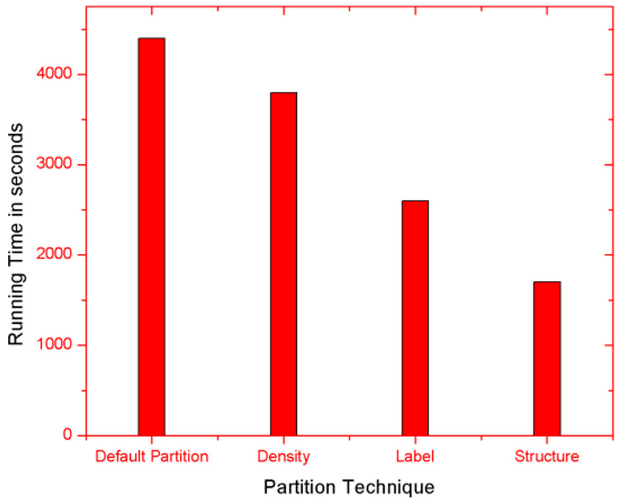


Figure 13. Query processing time versus graph partition technique.

Table 8
Database updates time.

Operation	Insert	Query processing	Both
Total update time (seconds)	70	60	80

5.2.11. Evaluation on database updates using real datasets

In this experiment, we show that in addition to efficient index construction and fast query processing, our index also has a very low maintenance cost. We consider three different scenarios of updates: insertion, query processing only and both. We use yeast dataset for this experiment which is shown in Table 5. For insertion only, we start with a database of 60 K graphs and insert another 10 K graphs. For both we use database of 60 K graphs and randomly choose to insert graphs into it from another 10 K graphs and 100 query graphs. The final database size we obtain at the end of all updates in each case 60 K. This experiment we conducted on 9 node cluster machine. The running time is recording and is shown in Table 8. From this experiment we came to know that both insert and query processing can be done simultaneously with same computational complexity and IO complexity.

6. Conclusions

In this paper we presented distributed methods to process multiple graph queries at a time instead of processing single query graph. This approach makes optimum utilizing of resources. We proposed two techniques to partition the data graphs based on labels of graphs and structure of graphs. Structure based approach is more efficient for query processing on large graph datasets. Efficient ways of Creation of Integrated Graph Index is reducing the IG size compared to normal of Integrated Graph Index creation. Structure based partition reduces the communication cost and computation cost for query processing even though it takes two mapreduce rounds. This MSP is suitable for graph database where commonality of graphs is more. In future we want to demonstrate how frequent subgraph mining can get benefits from graph indexing and we want to implement using in-memory distributed framework Spark.

References

Angles, Renzo, 2012. A comparison of current graph database models. In: Data Engineering Workshops (ICDEW) (Ed.), IEEE 28th International Conference. IEEE, pp. 171–177.

Aridhi, Sabeur, d'Orazio, Laurent, Maddouri, Mondher, Mephu Nguifo, En-gelbert, 2015. Density-based data partitioning strategy to approximate large-scale subgraph mining. Inform. Syst. 48, 213–223.

Chen, Chen, Yan, Xifeng, Yu, Philip S., Han, Jiawei, Zhang, Dong-Qing, Gu, Xiaohui, 2007. Towards graph containment search and indexing, Proceedings of the 33rd international conference on Very large data bases. VLDB Endowment, pp. 926–937.

Cheng, James, Ke, Yiping, Wai-Chee Fu, Ada, Xu Yu, Jeffrey, 2011. Fast graph query processing with a low-cost index. VLDB J. 20 (4), 521–539.

Cheng, James, Ke, Yiping, Ng, Wilfred, Lu, An, 2007. Fg-index: towards verification-free query processing on graph databases, Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. ACM, pp. 857–872.

Dean, Jeffrey, Ghemawat, Sanjay, 2008. Mapreduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113.

Euripides, G., Petrakis, M., Faloutsos, A., 1997. Similarity searching in medical image databases. IEEE Trans. Knowl. Data Eng. 9 (3), 435–447.

Ghemawat, Sanjay, Gobioff, Howard, Leung, Shun-Tak, 2003. The google file system. ACM SIGOPS operating systems review, vol. 37. ACM, pp. 29–43.

Giugno, Rosalba, Shasha, Dennis, 2002. Graphgrep: a fast and universal method for querying graphs, Proceedings 16th International Conference. Pattern Recognition, vol. 2. IEEE, pp. 112–115.

hadoop. http://hadoop.apache.org. [ ].

hdfs. http://hadoop.apache.org/hdfs/. [ ].

He, Huahai, Singh, Ambuj K., 2006. Closure-tree: an index structure for graph queries. In: Data Engineering, 2006. ICDE'06, Proceedings of the 22nd International Conference. IEEE. 38-38.

Haoliang Jiang, Haixun Wang, Philip S Yu, and Shuigeng Zhou. Gstring. A novel approach for efficient search in graph databases. In Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on, pages 566–575. IEEE, 2007.

Kang, U., Charalampos, Tsourakakis, Ana Paula, Appel, Faloutsos, Christos, Leskovec, Jure, 2008. Hadi: fast diameter estimation and mining in massive graphs with hadoop. ACM Trans. Knowledge Discovery from Data (TKDD) 5 (2), 8.

Kang, U., Tsourakakis, Charalampos E., Faloutsos, Christos, 2011. Pegasus: mining peta-scale graphs. Knowl. Inf. Syst. 27 (2), 303–325.

Song-Hyon Kim, Kyong-Ha Lee, Hyebong Choi, and Yoon-Joon Lee. Parallel processing of multiple graph queries using mapreduce. In The fifth international conference on advances in databases, knowledge, and data applications (DBKDA 2013), pages 33–38. Cite- seer, 2013.

Kuramochi, Michihiro, Karypis, George, 2005. Finding frequent patterns in a large sparse graph∗. Data Min. Knowl. Disc. 11 (3), 243–271.

Lubiw, Anna, 1981. Some np-complete problems similar to graph iso-morphism. SIAM J. Comput. 10 (1), 11–21.

Yifeng Luo, Jihong Guan, and Shuigeng Zhou. Towards efficient subgraph search in cloud computing environments. In Database Systems for Advanced Applications, pages 2–13. Springer, 2011.

Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel. A system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages 135–146. ACM, 2010.

Mansurul A Bhuiyan and Mohammad Al Hasan. Mirage: an iterative mapreduce based frequent subgraph mining algorithm. arXiv preprint arXiv:1307.5894, 2013.

Dennis Shasha, Jason TL Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 39–52. ACM, 2002.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, pages 1–10. IEEE, 2010.

Willett, Peter, Barnard, John M., Downs, Geoffrey M., 1998. Chemical similarity searching. J. Chem. Inf. Comput. Sci. 38 (6), 983–996.

Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on, pages 721–724. IEEE, 2002.

Xifeng Yan and Jiawei Han. Closegraph: mining closed frequent graph patterns. In Proceedings of the ninth ACM SIGKDD inter- national conference on Knowledge discovery and data mining, pages 286–295. ACM, 2003.

Xifeng Yan, Philip S Yu, and Jiawei Han. Graph indexing based on discriminative frequent structure analysis. ACM Trans. Database Syst. (TODS), 30(4):960–993, 2005.

Shijie Zhang, Meng Hu, and Jiong Yang. Treepi: a novel graph indexing method. In Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on, pages 966–975. IEEE, 2007.

Peixiang Zhao, Jeffrey Xu Yu, and Philip S Yu. Graph indexing: tree+ delta¡= graph. In Proceedings of the 33rd international conference on Very large data bases, pages 938–949. VLDB Endowment, 2007.

Zou, Lei, Chen, Lei, Xu Yu, Jeffrey, Lu, Yansheng, 2008. A novel spectral coding in a large graph database, Proceedings of the 11th international conference on Extending database technology: Advances in database technology. ACM, pp. 181–192.