Distributed Graph Indexing and Query Processing Using Map-Reduce

Fathimabi Shaik

ABSTRACT--- In recent times, we are observing that the of the size of the graph data is increasing and we cannot able to process by using a single machine in less time. In a distributed environment many users are giving the graph queries to get required data from large graph database. It becomes hard to get relevant graph data from a huge graph database. This paper address the issue of processing hundreds of query graphs from a huge graph database using distributed computing framework like Map-Reduce. We design a method to solve the problem of multiple graph query processing using inverted edge index and index maintenance. We develop a DIstributed Graph Indexing and Multiple Graph Query Processing Algorithm called DIGIMAP. DIGIMAP uses Replicated Join technique of Map-Reduce to filter the graphs and to do index maintenance. We did experiments using real-world graph datasets shows this approach improves the performance and quick processing of multiple graph queries over big dataset of graphs.

Keywords: graph query; graph database; big data; parallel processing; Map-Reduce; distributed graph query processing; Join technique

1. INTRODUCTION

Now-a-days many applications were using graphs to model the problem. Graph represents the relations among entities, each entity is a vertex their relationship is like an edge. Web data, social network graphs, there applications are using graph data. So, there is a great need of graph data management and mining tools to manage, process and analyze graph data efficiently.

The following is the classification of graph data mining:

Structural graph pattern mining: It finds the patterns from the graph database.

Indexing and Search of Graph Data: It prepares the index of all the grraphs in the graph database and search the graphs using exact approach or approximate approach in a big graph databases.

The categories of graphs available in the world are two, they are transaction graph setting [2] and a single big graph [3]. In the transaction graph setting, it contains a large number of relatively smaller graphs in the graph database, the other one in the singe-graph setting, the data is represented as a single graph. This work focuses on the transaction graph setting which consisting of thousands of graphs. Chemistry and bioinformatics domains there Transaction graph databases are used.

The graph queries are classified in to two categories based on the literature, they are sub-graph query and super-graph query.

Sub-graph query processing retrieves the matched graphs in the input graph database such that a given query graph is a sub-graph of them. Next, super-graph query retrieves all the graphs in the database such that the query graph is a super-graph of them.

Some researchers proposed methods to process the graph queries using graph index in a centralized approach such as GraphGrep[5], GraphIndex[6], FGIndex[7], Closure-Tree[8] etc. The main assumption in the above algorithms is that the graph data size is small and processed by using main memory, but is not correct. For example, SCI Finder, the collection of world's largest chemistry and related science information, says that every data for about four thousand new were added. For these kinds of applications, this approach is not suitable these kind of in memory approaches, as it is somewhat difficult to prepare and update the index and perform query processing on a centralized machine efficiently.

Big data is an emerging area in recent years, the number of researchers in different domains including social networks, chem.-informatics, knowledge disovery from data of all different types such as graphs. Map-Reduce based distributed solution was not popular that much. Since, Map-Reduce became the de-facto approach for processing large volume of data, the algorithm which takes less time for processing of Graph Queries is having high demand.

This paper proposed a DIGIMAP which a graph indexing and query processing using an open source implementation of Map-Reduce. First the naive method using all pair-wise subgraph isomorphism tests between a graph query set and input graphs was discussed. This method takes more time to process. This work introduces filter and verify approach in order to minimizes the subgraph isomorphism tests. DIGIMAP partitions the graphs into a set of machines. Collectively all machines generate the Inverted Edge Index. This Index is stored in the HDFS. Filter and verify steps are used when a set of graph queries are being processed. According to our knowledge concern, this is the first work which process multiple graph queries over a large graph dataset using Join technique in Map-Reduce.

We claim the following contributions:

- Filter the graphs using Replicated Join in hadoop.
- Multiple graph queries can be processed as batch processing.

The organization of this paper is: The existing works were discussed in second section. The preliminaries were presented in third section 3. The proposed method was discussed in fourth section. We provide the results of the experiments in fifth Section. The conclusion of the paper was presented in sixth section.

Revised Manuscript Received on February 11 , 2019.
Fathimabi Shaik, (skf_cse@vignanuniversity.org)

2. RELATED WORK

Graph indexing techniques are of two types: path-based Index, sub-graph based Index. In Path-based Index, generate all the paths upto some maximum length and then build index for all the paths. For example graphgrep [5], SubGraph based index generates all subgraphs and then index the subgraphs. For example gIndex [6], FGIndex [7], Closure-Tree [8], GString [9]. These algorithms are doing complex operations to get paths of all lengths or mine frequent subgraphs which are used for indexing. This is a time taking process. The main assumption of the above algorithms is that the size of the graph database size is less and it takes less time to process the queries using an in-memory based method. Processing large graph dataset using a single machine is very hard. For example the PubChem project stores more than 30 million chemical compounds, and the storage requirement is more than tens of terabytes [11]. In distributed environment heterogeneous types of graphs are collected and stored to process. When the dataset is large, mining the large graph dataset and then indexing is a complex work. To overcome this drawback, in this paper we used edge based index because edge is the basic unit for any graph.

Both industry and academia are extensively studying the Map-Reduce paradigm and designing [5]. The distributed method was presented by Map-Reduce in order to process the big data with no hassle of managing the jobs across nodes. Map-Reduce is following the paradigm of "Moving Code to the Data" big data processing using cluster. Apart from this, it uses a HDFS (Hadoop Distributed File System), optimized for IO performance to process big data. The another reason why Map-Reduce was used by main researchers is that it hides the lower level details from the programmers and there, by they concentrate on the algorithm to solve the problem

2.1 Join data from different sources using Map-Reduce

We have different approaches to join data in Map-Reduce. The comparison of different join algorithms in Map-Reduce is presented in [18], in that most popularly used join techniques are as follows.

Reduce-side join

The other name for this Reduce-side join is Repartitioned Sort-Merge Join. Each record is associated with tag that is data source name. Most of the processing is done at the reduce side. In Hadoop a contrib package called data join acting as framework to join data in Hadoop. It is a flexible technique, but it is an inefficient technique. The data join operation does not perform until the reduce phase. All the data must be shuffled across the network and the most of the data was dropped. If the map phase eliminates the unnecessary data then this work is the efficient one. The total process of join work is done by the map phase.

Replicated join using Distributed Cache: Most of the applications where we are joining big data one source is big and other one is small which can be fit in memory of mapper. If we place this small data to all mappers then the efficiency is more. It is also called as Replicated join in the domain of database, because one data table is replicated across the machines in the cluster.

Distributed Cache: This is one of the best mechanism available in Hadoop to distribute files to all machiness in a cluster. It distributes small data to all mappers. So join takes place in the mapper. So it reduces the communication cost.

Song-Hyon Kim et al.[10] proposed Map-Reduce based multiple graph query processing with help of Bloom filter. Here Bloom Filter is used to filter the data graphs based on edge label. It takes lot of time to reduce the candidate data graphs based on the query graphs using Bloom filter and more over it can produce false positives. It is using naive approach. Y Luo et al.[14] designed an algorithm to perform subgraph search in cloud environment, which process single query using two index files. This work filters the graphs only. Large scale graph processing was presented in [15] and [16], these works processed a single large graph rather than a number of small graphs for example social graph etc.

2.3.1 PRELIMINARIES

Definitions

Graph A graph is a collection of vertices V, a set of edges E such that E⊆ V x V, the labels of vertices or edges is denoted by L and a mapping M: V ∪ E □ L, denoted by a tuple g = (V, E, L, M). Vertices can also be represented as V(g) and edges as E(g).

Subgraph Isomorphism For the two graphs g = (V, E, L, M) and $g^1 = (V^1, E^1, L^1, M^1)$, g is subgraph isomorphic to g^1 denoted $g \subseteq g^1$ if and only if there exists an injective function. The function is injective means there is a one-to-one relation, if every element of the co-domain is mapped to by at most one element of the domain. An injective function is an injection. $\forall x, y \in A f(x) = f(y) \Rightarrow x = y$ or Or, equivalently (using logical transposition) $\forall x, y \in A, x \neq y \Rightarrow f(x) \neq f(y)$

Problem Statement

Problem Statement: Let D is a graph databsae which consists of set of graphs such as g_1, g_2 , $g_3 ... g_n$ and a set of query graphs q_1, q_2, $q_3 ... q_m$ be a graph query set represented as Q, such that | Q |<< |D|. We are finding all the data graphs where the given query graphs are subgraph isomorphic.

This work processes a stream of queries (hundreds) at a time. A batch of queries are coming from number of sources in distributed environment. This work processes stream of queries that means they are coming in a high speed. In distributed environment most of the applications require the the quick response. Moreover, processing the queries as batches eliminates the repetitive work work of processing the common parts of the all the queries, this will improve the throughput.

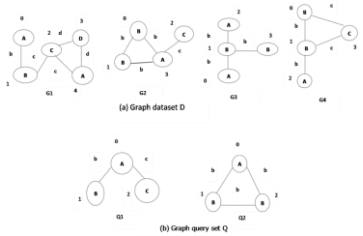


Fig. 1: A graph database and few query graphs

3.2 GRAPH REPRESENTATION

Single line representation of a graph is proposed in [10].

A Single Line representation of a graph: Generally Graph dataset is obtained in a format of multiple-line. Some existing hadoop based graph data mining works such as frequent subgraph mining work[2] splits the graphs into a set of files in the data preparation step and the resultant files are gives as input to Map-Reduce program. This works uses a single-line layout. In a single-line layout there exists sequential representation of graph g as vertices and edges in g, i.e {| V(g) | , | E(g) | , l(V(g)), E(g)} where e ϵ E(g) is represented as from-gid, to-gid, Graph is represented as graph number a unique identifier for a single graph, the number of vertices, the number of edges, Labels of all vertices, and edge list.

For example: g2,4,4,A,B,C,E,0,1,b,0,2,d,1,2,e,2,3,f.

Here g2: graph id

4 : the count of vertices

4: the count of edges

A,B,C,E : vertice labels.

0,1,b : source vertice id , destination vertex id and edge label.

4. PROPOSED APPROACHES

First we talk about a naive approach they use Map-Reduce to process multiple graph queries. Later, we present the approach of our DIGIMAP.

Naive Approach

Naive approach processes the multiple graph queries by performing the subgraph isomorphism test on every pair of data graph and a query graph. The graph dataset was partitioned into number of splits and stored by sevmachines in the cluster. Query Graph Set is available to all machines by placing it in Distributed Cache. Each machine do the pair wise subgraph isomorphism test and if it is yes then it sends it to the reducer. At the reducer we get the query id and list of matched data graph ids. This approach performs the number of subgraph isomorphism test equal to |Q| x |D|. It is taking more time, to overcome this we propose the DIGIMAP

Introduction to DIGIMAP

DIGIMAP is processing multiple graph queries using Replicated join using Distributed Cache. It is using the approach which has filter phase followed by verify phase which reducesthe subgraph isomorphism tests. According to the filter and verify approaches, the candidate graphs for the given query graphs are filtered then test subgraph isomorphism with only the candidate graphs, instead of testing subgraph isomorphism with all the graphs in the input graph database. To overcome this subgraph isomorphism tests, this work filters irrelevant graphs by checking the features of query graph with the set of features of data graph.

This filtering approach reduces the overall execution time significantly since it excludes many graphs in advance so that the number of graphs to be verified is quite reduced.

The basic steps of DIGIMAP are

- Build the Inverted Edge based Index / Inverted Edge Occurrence Index for large graph database.
- Reduce the subgraph isomorphism tests with help of Filter phase.

Index maintenance.

The work completes the work in three phases: Index Building, Multiple Sub-graph query processing phase and Index maintenance phase. Figure 2 shows the index building and index maintenance phases. Figure 2 shows that one map reduce round is used for index creation and one round is used for index maintenance. Figure 3 illustrates the overview of the query processing phase. One Map-Reduce round is used for filter and one round is used for verify.

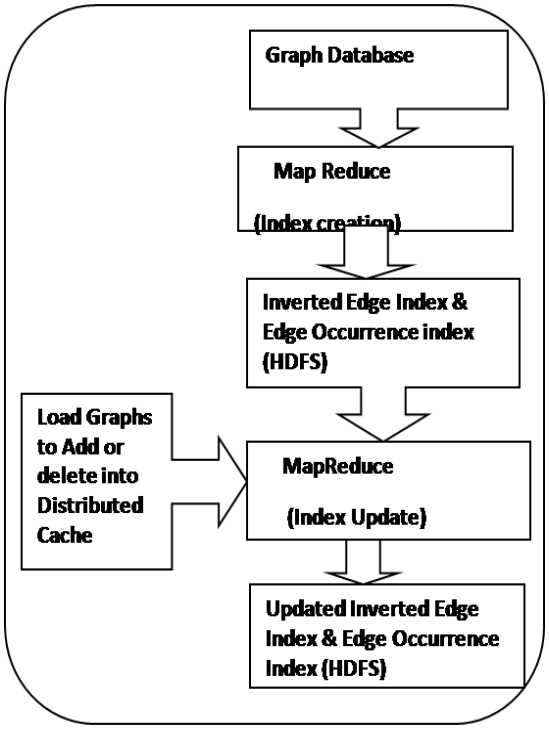


Fig. 2: An overview of Index Building and Index Maintenance

4.1 Building Inverted Edge Index

In this phase we build edge based inverted index by using one Map-Reduce round. It takes graph database file(s) as input and builds inverted indexes

Mapper Output

Edge	Graph id
<A B b>	G1
<B B b>	G1
<A C e>	G1
<C D d>	G1
<A D d>	G1
<B B b>	G2
<A B b>	G2
<A C e>	G2
<A B b>	G3
<B B b>	G3
<A B b>	G4
<B B b>	G4
<B C e>	G4

Reduce

Edge
< A B b >
<B B b >
<A C e >
<B C e>
<C D d>
<A D d>

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

Figure 3(a) Inverted Edge Index

Edge	Query set
<A B b >	Q1, Q2
<A C c >	Q1
<B B b>	Q2

Figure 3(b) Inverted Edge Index for Query graphs in a Preprocessing step

Figure 3 Inverted Edge Index created for graph database and query graphs for all the edges. With one Map-Reduce round the inverted edge index is built. Inverted Edge Index File Format Edge - List of graph ids

A B c - g1, g2, g3.

Inverted Edge Index : All the graphs in graph database file(s) as input and gives inverted index over the edge set as inverted edge index file. It performs two steps.

- Parse the graph: Divide the graph into edges.
- Find each edge and its graph ids list:

Each mapper sends edge and its graph id. Instead of sending each edge and its graph id to the reducer, here we are using combiner to get local Graph ids of a particular edge which is reducing the data flow in the network. No need to write separate class for combiner. The reducer class is used as combiner. The algorithm 1 shows the mapper and reducer procedures to create Inverted Edge Index. In this phase Map-Reduce job

Algorithm 1: Inverted Edge Index Building

```
Result: Inverted Edge Index
Data: Large graph database GD
1  Class Mapper
2  method map (N:Offset , V:[gid,gcode])
3  for each graph with Gi in graph database GD do
4  for each edge labeled with EdgeLabeli in V do
5  emit (EdgeLabeli,Gi)
6  Class Reducer
7  method reduce (Edge Label e, V: List of gids)
8  for each edge e in V do
9  concatenate all graph ids to generate
   GraphSet Strng GSi : Gi1, Gi2 , Gi3
10 emit (EdgeLabeli,GSi)
```

takes the input graphs and builds inverted indexes over the edge set. In the input file each line is a graph. Each record of the input split is parsed by the Map function and it sends Edge and Graph number as output to the reducer. similar edegs are going to the one reducer because of sort and shuffle phase. In the reduce function all the similar edges are aggregated together, and the reducer function generates the output file <Edge and set of graph nos> as Index. Figure 3 (a) shows the mapper output and reducer output for the database shown in Figure 1.

Filter Step using Inverted Edge Index

Preprocessing Step for Filter: Inverted Edge Index for query graphs: In this preprocessing step we prepare the Inverted Edge Index for query graphs using single machine. If we do this work in distributed environment, each machine has to do same work that means we are wasting the time and resources for query processing. Instead of loading query graphs into Distributed Cache, we are loading inverted edge index of query graphs. The result is this preprocessing step is shown in the Figure 3 (b). Filter step is reading Inverted Edge Index file from HDFS and Query edge index file is loaded into distributed cache. Query Edge Index is available to all machines. The Input Index file is partitioned into blocks of equal size and each of the blocks is assigned to a mapper at map stage. Edges and graph ids are comming from the input ,do join operation if edges is present in query index and the output is sent to reducer . In our approach, filter is performed by using join operation based on edge in the mapper and intersection operation in the reduce phase. Algorithm 3 shows the filter step.

Figure 4 An overview of DIGIMAP Graph Query Processing

The following are the steps in Filter:

- Load the Inverted Edge Index of query graphs into distributed cache.
- Read Inverted Edge Index from HDFS as input.

Do join operation based on each edge label of query graphs.

Table 1 Join Step Result

Table 1 shows the result of join step.

Reducer received every query id and its set of graphs ids as value. Then reducer is doing intersection operation. The result after intersection is shown in table 2.

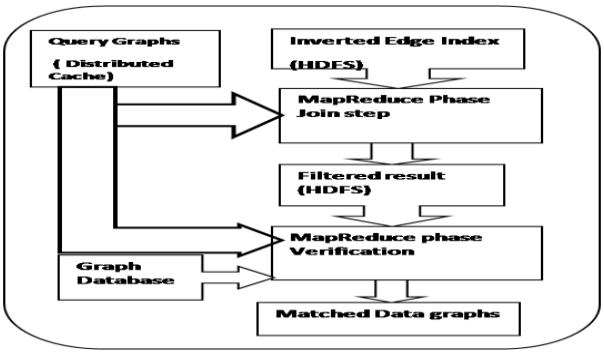
Query Graph Id	Data Graphs
Q1	G1,G2,G3,G4
Q2	G1,G2,G3,G4
Q1	G1, G2
Q2	G2,G3,G4

Tabel 1: Join Step Result

Query Graph Id	Ids of Data Graphs
Q1	G1 G2
Q2	G2 G3 G4

Table 2: Result after Intersection

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication



Algorithm 2: Filter using Inverted Edge Index

Result: query id - graph ids
Data: Inverted Edge Index of graph Database D , Inverted Edge Index of query graphs is loaded into distributed cache

```
1  class Mapper
2  method Setup
3  read query inverted edge index from distributed cache
   and prepare hashtable as queryindex;
4  end
5  method Map (key: offset, value: edge label, graph ids
   list)
6  parse the value;
7  get edge from value;
8  get graph ids from value into gids;
9  search edge in queryindex;
10 if edge found in queryindex then
11 gids loaded with graph ids list;
12 qids loaded with query ids list from hash table;
13 for each qid in qids do
14 emit (qid, gids)
15 end
16 end
17 class Reducer
18 method Reduce (Key qid, value: list of gids list)
19 get qid and its gidslist;
20 result=intersect the gids list;
21 emit (qid, result);
```

Verification Step (Subgraph isomorphism test)

Preprocessing Step for Verification: Format change of Filter Result

The result of filter step is shown in Table 2 This format is not suitable for verification round. We convert this result into the required format as preprocessing step and is shown in Table 3.

The following are the steps in Verification

- Tthe result of filter round and query graphs is loaded into Distributed cache
- Read Graph Database as Input
- Do the subgraph isomorphism check using depth first traversal of query graph

Graph id	List of query graph ids
g1	q1,q2
g2	q1
g3	q2
g4	q2

Table 3: The Result of Filter step after Format Change

Display the matched query id and its graph ids.

Query Graph Id	Ids of Data Graphs
Q1	G2
Q2	G2

Table 4: Result after Verification

In this step subgraph isomorphism test has done by checking the survival of depth first traversal edge order of query graph in the data graph. This approach is taking less time compared to dfscode verification used in gspan[4]. In gspan we find minimum depth first search code and according to the minimum dfs code we compare. Here we are verifying subgraph only based on the checking of depth first order of query graph in data graph. It takes less time. Only Map function is enough for the verification and it sends the result to HDFS. The Algorithm 3 shows the verification process. The final result after verification is shown in Table 4.

Algorithm 3: Verification Step using original Graph Database

Result: final result : query id - graph ids
Data: Graph Database, Filter result and query graphs are loaded into Distributed Cache

```
1  class Mapper
2  method Setup
3  The result of filter step after format change is loaded
   into hadoop distributed cache
4  qlist=read query graphs from distributed cache
   filter=read filter step result line by line divide filter
   into gid, qidlist
5  graphtable.put (gid, qidlist)
6  end
7  method map (Key: offset, Value: gid, gcode)
8  parse Value and get gid and gcode
9  check gid in graph table
10 if found then
11 get qidlist;
12 for each qid in qidlist read query qid from query list;
13 if subgraph isotest (query, gcode) then
14 emit (qid, gid);
15
16 end
```

```
17 subgraph isotest(query,graph)
18 begin
19 prepare edgelist according to depth first traversal of
   query graph ;
20 start from the first edge in edgelist get depth first
   traversal in data graph based on the query depth first
   traversal edge order
21 if found
22 return TRUE;
23 else
24 return FALSE;
25 end
   end
```

4.4 Index maintenance phase

When we want to add or delete some graphs from database we need to update the inverted edge index. In this phase the addition of new graphs to the index and deletion of graphs from the index are performed in a single Map-Reduce round.

Preprocessing step: For all the graphs which we are going to add / delete from the index prepare inverted edge index with attributes I – insert and D - Delete.

The index is shown in Table 5 for the graphs shown in Figure 5.

Each machine read the update inverted edge index from distributed cache. Index file is the input to the mappers. Each mapper read the contents of inverted edge index file line by line and verify with update inverted edge index using edge label, if it is matched then do the modification and send the modified copy to output file ie stored on hadoop distributed file system (HDFS). The overview of this phase is shown in figure **1.**

This index maintenance type is easy and requires less time instead of reading edge contains in a particular file[14]. concurrently all machines are involved in the updating process. The algorithm for index maintenance is shown in Algorithm 4.

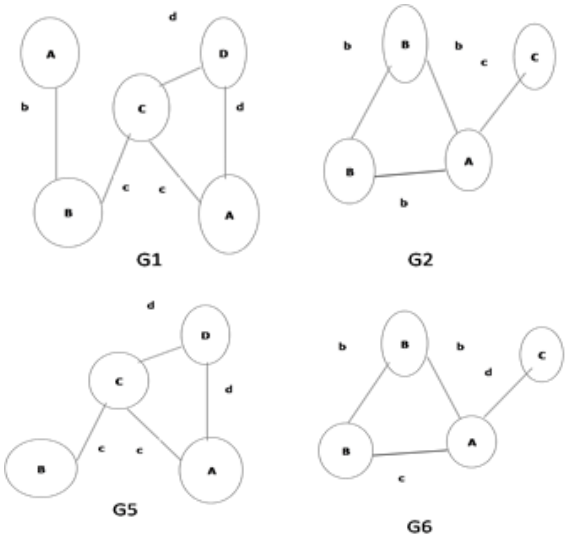


Figure 5 Graphs to insert into / delete from the Graph database

Algorithm 4: Inverted Edge Index Maintenance Algorithm

Result: Updated Inverted Edge Index
Data: Inverted Edge Index
Class Mapper
Method Initialize
Load inverted edge index prepared for the graphs to update in the
index into distributed cache
prepare hash map for this index as HE

Method map(N:Offset , V:[edge, graph set])
Get edge into E
Get graph set list into GS
Join E with edge in HE
If (exists)
{
Get update list from HE into UPDATELIST
Do changes in GS according to the UPDATELIST
}
Emit (edge,GS)

5. OPTIMIZATION

We discuss one optimization technique for the filter step of query processing in this section.

5.1 Count Information in the Index

Along with graphid the count (how many times the edge present in this graph) is added in the index file. For the above example <A B c> g1(2),g2 If number is not specified after graph id means edge exists one time in that graph, otherwise the specified no of times. During filter step count is used to filter the data graphs. If the count in data graph is more than the count in the query graph then only it will come into result. This improves the performance of filtering more. The filtering rate is more. The performance variation is discussed in the result section.

Edge	Graph Set
< A B c >	G6 I
<B B b>	G6 I, G2 D
<A C c>	G5 I , G1 D, G2 D
<B C c>	G5 I, G1 D
<C D d>	G5 I, G1 D
<A D d>	G5 I, G1 D
<A C d>	G6 I
<A B b>	G6 I, G1 D, G2 D

Table 5. Inverted Edge Index for the graphs to insert and delete

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

5.2 Using In-Mapper Combiner in the Index building phase

In-Mapper combiner: In-mapper combiner is a Instead of sending each edge and its graph id (count) information every time for each appearance of that edge we are using in-mapper combiner. In the mapper function an associative array is used inside to concatenate graph ids for a particular edge within split of the split edge in the split, this version emits a key-value pair for each unique edge in the split. This results reduces the number of keys transfored over the network—from the order of total number of edges in the graphs to the order of the number of unique edges in the graphs. After processing the total input the result is emitted in the close method. The Algorithm for Index Building using in-mapper combiner is shown in Algorithm 5.

Algorithm 5: Inverted Edge Index Building using In-mapper combiner

Result: Inverted Edge Index
Data: Large graph database D
Class Mapper
Method Initialize
H ← new Associative Array

Method map(N:Offset , V:[gid, gcode])
for each graph labeled with Gi in graph database D **do**
for each edge labeled with Edge Labeli in V **do**
if(Edge Labeli exists in H)
H{EdgeLabel}← H{Edge Label} + Gi
Else
H.put(Edge Labeli,Gi)

Method Close
for all edges EdgeLabel ∈ H **do**
return (EdgeLabel, H{EdgeLabel})

6. EXPERIMENTS

The performance of DIGIMAP using real and synthetic datasets was presented in this section. This work focuses on the improvement of efficiency in Map- Reduce. The query processing in fourth section, in terms of processing time consumed to each stage of the query processing. It first describes the used datasets and implemen- tation details. At last it includes the analysis of results.

6.1 Experimental setup

6.1.1 Datasets

This work uses the datasets and were shown in table 6. They were acquired from online source that has graphs extracted from PubChem website. It has large number of chemical structures. Each graph has vertices 24.99, edges are 26.65, 2.5 distinct labels for vertices, the average distinct labels of edges are 2.1, the total unique vertex labels and edge labels are 80 and 4, respectively. Its size is 240MB. Along with the existing graph datasets, we generated the three lakh graphs. Graph queries are generated randomly and has tens, hundreds , thousand o queries.

Graph Dataset	Number of Graphs	Average size of each graph
Yeast	79599	23.5
P388	41470	26
SN12C	40002	31.5
OVCAR-8	40514	31.5
NCI-H23	40351	31.5
MOLT-4	39763	30.5
PC-3	27507	32
SF-295	40269	32
SW-620	40530	31

6.1.2 Implementation platform

We implement this work in java language and we use Hadoop (version 1.2.1) an open source version of MapRe-duce. The database files are stored in the HDDFS whose origin is Google File System GFS[[17]. The experiments of our approach were carried out on eight node cluster.

Namenode: HP Proliant DL380P Gen8 2 x Intel xeon CPU E5-2640 (2.5 GHz / 6-core/15MB / 95w) Processor Intel 7500 chip set with node controller, 64 GB RAM, HP SA 410i RAID controller with 1 GB Storage: HP MSA2040 SAN SFF / 24 x 300GB HDD / 8*16GB POETS OS: Rocks Cluster 6.1.1 + CentOS 6.5 Server with Hadoop1.2.1

NData node: Intel xeon CPU E5-2640 (2.5 GHz / 6-core/15MB / 95w) Processor, 16GB RAM, 2* 300GB HDD Intel 7500 chip set with node controller

6.2 Experimental results

6.2.1 Time required to build the index verssus the Number of Data Graphs

Figure 6 shows how the time taken by our approach for different number of data graphs. We observe that the edge indexing is taking less time when we increase graphs this is because it is distributing the work among nodes. After some time, number of graphs are increasing time is also increasing. Figure 7 shows the time taken to create index for different real datasets.

6.2.2 Filter time for different number of data graphs

To find how much time is taking for filter data graphs fordifferent number of data graphs. Figure 8 shows that Edge Indexing is taking less time for filtering even we increase data graphs because the work is distributed to all the nodes. Figure 9 shows the comparison for both indexing techniques we came to know that when we used Inverted Edge Index with count it is taking more time. But it filter mores graphs.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

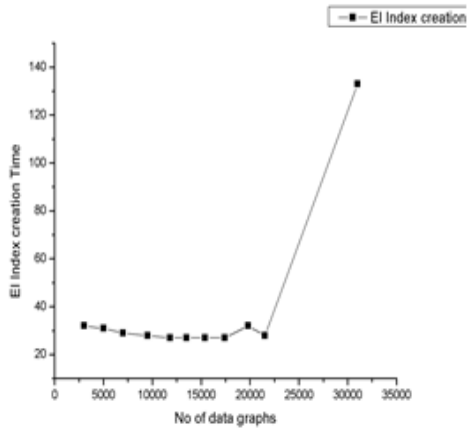


Figure 6 Inverted Edge Index Building Time

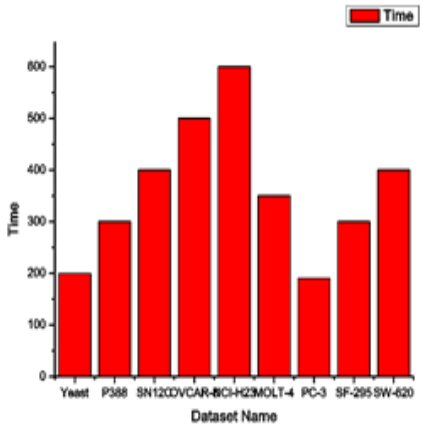


Figure 7 Inverted Edge Index Building Time for Real Datasets

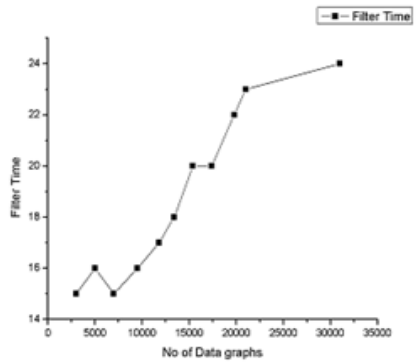


Figure 8 Filter Time

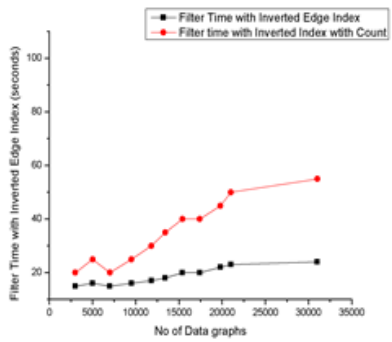


Figure 9 Comparison of Filter Time

6.2.3 Query Verification Time using EI and EI with Count

In this experiment we conducted query processing using both EI and EI with count index. Using EI with count is filtering more no of candidate graphs. So the verification time is less compared to EI index. Figure 10 shows the comparison.

6.2.4 Query Processing Time time for Synthetic data sets

In this experiment we used 3 lakhs data graphs and different number of query graphs. Figure 11 shows that as no of query graphs are increasing then time is increasing. Based on query graphs, if query graphs are not matched with data graphs then it takes less time because in the filter itself we eliminated false positives.

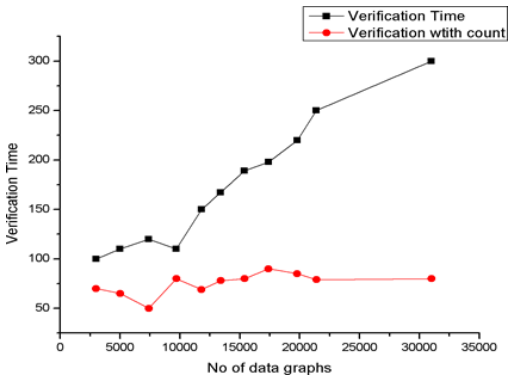
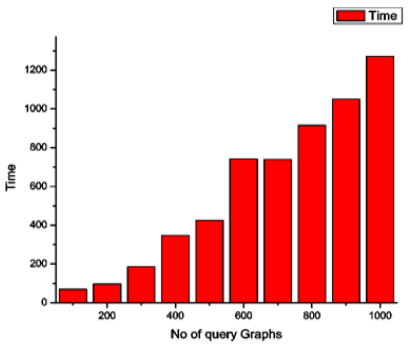
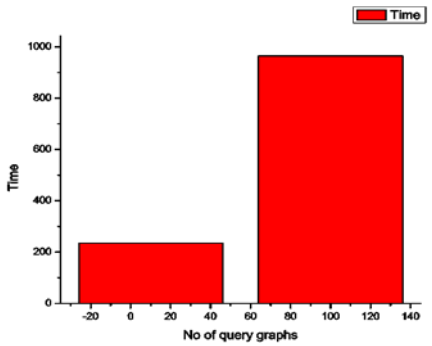


Figure 10. Verification Time



(a) Query Processing Time for synthetic data sets



(b) Query Processing Time for Yeast data sets

Figure 11. Query Processing Time for Synthetic and real data sets

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

6.2.5 Index Maintenance Time vs no of graphs to add/delete

In this experiment we used 3 lakhs data graphs and different number of graphs to update in the index. Figure 12 shows that as no of graphs are increasing sometimes time is not increasing that means the load is distributed to all machines.

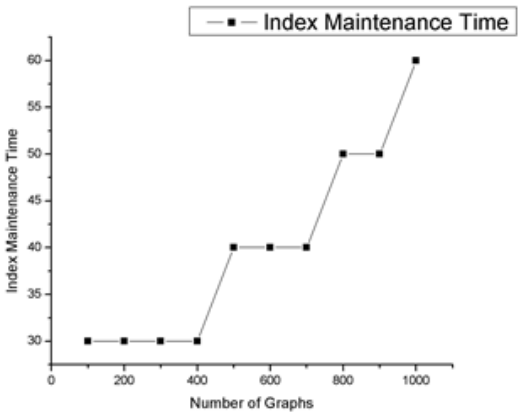


Figure 12 Index Maintenance Time

6.3 Conclusions

In this paper we present multiple graph query processing using Join Technique in Map-Reduce. Two indexing methods are proposed and used for batch graph query processing using Map-Reduce. This work shows that the performance of DIGIMAP for both real and synthetic datasets and for number of query graphs. We also compare the filtering rate after optimization. Using DIGIMAP we can process hundreds of queries and we can do index maintenance in less time. In future, we will develop a method which demonstrate the use of graph index to do frequent subgraph mining over transaction graph databases.

REFERENCES

1. Aggarwal, C.C., Wang, H(eds): Managing and Mining graph data. Kluwer Academic Publishers, Dordrecht (2010)
2. Mansurul A Bhuiyan, Mohammad AI Hasan : MIRAGE An Iter- ative Map Reduce based Frequent Subgraph Mining Algorithm 2013.
3. M. Kuramochi and G. Karypis, Finding frequent patterns in a large sparse graph*, Data mining and knowledge discovery, vol. 11, no. 3, pp. 243271, 2005
4. X. Yan and J. Han.,"gSpan: Graph-Based Substructure Pattern Mining",Proc. 2002 of Int. Conf. on Data Mining (ICDM'02).
5. Giugno, R., Shasha, D: Graph Grep: A Fast and Universal Method for Querying Graphs. Proceedings of ICPR 2, 112-115 (2002)
6. Yan,X.,Yu,P.,Han, J.: Graph Indexing Based on Discriminative Frequent Structure Analysis. ACM Tranasactions on Database Systems 30(4),960-993(2005)
7. Cheng, J., Ke, Y., Ng, W., Lu,.: FG-Index: towards verification- free query processing on graph databases in: Proceedings of ICDE (2007)
8. He,H.,Singh, A.K.: Closure-Tree.: An Index Structure for Graph Queries. In Graphs, Proceedings of ICDE (2006)
9. Haoliang Jiang ; Haixun Wang ; Yu, P.S. Shuigeng Zhou, :GString: A Novel Approach for Efficient Search in Graph Databases Proceedings of ICDE (2007)
10. Song-Hyon Kim, Kyong-Ha Lee, Hyebong Choi and Yoon-Joon Lee, Parallel Processing of Multiple Graph Queries Using Map-Reduce,DBKDA,2013
11. Willet, P.: Chemical similarity searching. J.Chem. Info. Comput. Sci. 38,983-996(1998)
12. Dean, J., Ghemawat, S.: Map-Reduce: Simplified data processing on large cluster. In: Proceedings of OSDI, pp 137-150(2004)
13. James Cheng, Yiping Ke, Ada Wai-chee Fu, Jeffrey Xu Yu: Fast Graph Query Processing with a Low-Cost Index.VLDB 2011.
14. Yifeng Luo,Jihng Gun nd Shugeng Zhou, Towards Efficient SubgraphSearch in Cloud Computing Environments. Spinger-verlag Berlin Heidelberg 2011.
15. G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn,N. Leiser, and G. Czajkowski, Pregel: a system for large-scale graph processing, in Proceedings of the 2010 interna- tional conference on Management of data. ACM, 2010, pp.135146.
16. U. Kang, C. Tsourakakis, and C. Faloutsos, Pegasus: mining peta-scale graphs, Knowledge and information systems, vol. 27, no. 2, pp. 303325, 2011.
17. S. Ghemawat, H.Gobioff, S.T.Leung. The Google File System, in:proceedings of the 19th ACM Symposium on Operating Systems Principles, vol.37 of SOSP '03, ACM, New York, USA, 2003.
18. S. Blanas, J. Patel, V. Ercegovac, J. Rao, E. Shekita, and Y. Tian, A comparison of join algorithms for log processing in Map-Reduce, in Proceedings of the 2010 international conference on Management of data. ACM, 2010, pp. 975986.
19. F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating sub-graph instances using map-reduce. In ICDE, pages 6273, 2013
20. Hadoop. http://hadoop.apache.org.
21. Nci. http://cactus.nci.nih.gov/download/nci.
22. Pubchem. http://pubchem.ncbi.nlm.nih.gov.
23. Jimmy Lin and Chris Dyer: Data-Intensive Text Processing with Map-Reduce, 2010

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication