International Journal of Psychosocial Rehabilitation, Vol. 25, Issue 03, 2021
ISSN: 1475-7192

MapReduce based Distributed Graph Grep using Edge Occurrence Index

Fathimabi Shaik[1], Ebenezer Jangam[2]

Department of Information Technology

Velagapudi Ramakrishna Siddhartha Engineering College Vijayawada, India

fathimabi@vrsiddhartha.ac.in[1], jebenezer@vrsiddhartha.ac.in[2]

Abstract

Graph query processing is a very important application for the graph data. The graph data set size increases day by day due to digitization of all types of data, in order to process the large amount of graph data using number of machines not by single machine. Graph query processing using distributed frameworks like Hadoop is a challenging task. Many users are giving graph queries to process in distributed environment, in an interactive way it has to process all the queries. It becomes hard to process graph queries from a big graph dataset. This paper mainly emphasis on processing of multiple graph queries over a large set of graphs, using MapReduce framework. We introduced edge occurrence index to process multiple queries using filter and verify technique in MapReduce. We are using structure based graph partitioning to distribute all the graphs to the machines in the cluster based on structure of the graphs. The proposed algorithm is called as MapReduce based Distributed Graph Grep using Edge Occurrence Index MRDGG. Extensive experimental result analysis on various real-world graph datasets proved that the proposed work improves the performance and reduces the time for multiple graph query processing for massive collections of graphs.

Keywords: graph query; graph dataset; bigdata; parallel processing; MapReduce; Distributed graph query processing; Join technique.

1. INTRODUCTION

Graph is complex data structure to represent relations among the entities such as chemical compounds, interactions of protein and web data can be modeled by graphs in many applications [1]. Mining tools and graph data management are useful for an user to store, manage and analyze huge graph dataset efficiently. Graph data mining can be broadly categorized into the following:

Structural Pattern Mining: When a graph dataset is given, extracting the interesting structural patterns is referred to as the structural pattern mining.

Graph Indexing and Search: It focus mainly on the creation of graph index and then perform search operation. This searching can be done exactly in small graph databases. Approximately searching strategy can be applied on large graph data bases. Graph datasets are available in the following two categories: Graph transaction setting[2] dataset consists of a large number of small graphs. Single graph setting[3] the data is a single big graph. This paper mainly emphasis on the graph transaction setting which consists of millions of graphs. Scientific domains such as chemistry and bio-informatics are the major dependents of Transaction graph datasets.

Graph queries are used to retrieve the information from graph databases. These graph queries are categorized into 2 types. They are Subgraph query: Retrieving all the graphs in the huge graph dataset for which the given query graphs are the subgraphs. Supergraph query: Retrieving all the graphs in the dataset for which the given query graphs are the super graphs.

Many researchers proposed various methods to process graph queries using graph indexing such as GraphGrep[4], GraphIndex[5], FGIndex[6], Closure-Tree[7] etc. These algorithms assume is small and fit in main memory, but the size of the data is increasing day by day, for example, SCIFinder reports that about 4000 new compound structures are added each day. In such scenarios these kind of indexing algorithms are unfit. For these kind of applications, it is not appropriate to employ 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 Frameworks are used by the researchers in all the domains where the data is huge.

This work processes multiple graph queries using Hadoop MapReduce. At first we have to begin with a naive approach and its drawbacks. To overcome the time consumption drawback of naive approach and number of subgraph isomorphism tests we used filter and verify scheme. MR Distributed Graph-Grep partitions the graphs into set of machines. Collectively all machines generate the Inverted Edge Occurrence Index. This Index is stored in the HDFS. Filter and verify steps are used when a set of graph queries are being processed. This is the first work to generate inverted edge occurrence index for graph database and processing multiple graph queries over a large graph dataset using Join technique in MapReduce. The following are the contributions of this paper:

• We introduce inverted edge occurrence index and its maintenance.

• The graphs are filtered using Replicated Join using Distributed Cache in hadoop.

• Multiple graph queries can be processed as batch processing.

• MRDGG performance was demonstrated on synthetic as well as real world large datasets.

The rest of the paper is as follows: the related work was presented in Section 2 and preliminaries were presented in section 3. Proposed method was explained in Section 4. In Section 5 experimental results were presented. Finally, Section 6 concludes the work.

2. RELATED WORK

Graph indexing techniques are classified into 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[4]. SubGraph based index generates all subgraphs and then index the subgraphs. For example gIndex[5], FGIndex[6], Closure-Tree[7], GString[8]. Graph indexing algorithms assume that the data set is small and the mining task will be finished in a small amount of time by using an in-memory based methodology. Most of the graph data sets are difficult to handle within a single machine due to their size and complexity for example the PubChem project now a days serves more than 30 million chemical compounds, whose storage capacity hits terabytes[12] of memory. MSP[10] proposes structure based graph partitioning to process multiple subgraph query processing using MapReduce. In[11] Extended graph partition and enumeration of multiple subgraphs are presented.

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. MapReduce Technique has grabbed the attention of people from both of industry and academia[5]. MapReduce technique presents a distributed way to process data intensive jobs without having the difficulty in handling jobs across nodes. It opts for a data intensive approach of distributed computing with the ease of "moving computation to data". Apart from this, to improve the IO performance and to handle massive data distributed file system was utilized. Another interesting fact is that high level details are hidden from programmers which make them to develop distributed solutions is an easy way. Most of the people admired because of this reason.

2.1 Join data from different sources using MapReduce

There are several possible approaches with different tradeoffs. The comparison of different join algorithms in MapReduce is presented in [19].

• Reduce-side join:

It is also termed as re-partitioned sort-merge join. Each record is associated with tag that is data source name. Most of the processing is done at the reducer. Joining operation is carried out at the end

of reduce phase. At first all the data was shuffled across the network and then majority of the data was dropped during the joining process. The more efficient mechanism would be elimination of more data in the map phase.

• Replicated join using Distributed Cache

By using join operation we can join data from two sources, one source is big and the second source is small. A tremendous gain in efficiency is achieved if the source which is small fits in the memory of mapper and all mappers perform joining in the map phase. In database terminology it is called replicated join as one of the data table is replicated across all nodes in the cluster.

Distributed Cache: To distribute files to all nodes in a cluster, Hadoop has a new mechanism termed as distributed cache.

Using distributed cache small data is available to all mappers. So join takes place in the mapper. So it reduces the communication cost. Song-Hyon Kim et al.[9] proposed a parallel approach to process multiple graph queries using MapReduce and Bloom filter. Here BloomFilter is filtering the data graphs based on edge label. To filter the graphs using Bloom filter is a time consuming process and producing false positive values is another drawback of Bloom filter. It is using naive approach.

A solution to the subquery search in cloud was proposed by Y Luo et al.[15], which process single query using two index files. This work filters the graphs only. Big graph processing frameworks are Pergel [16], Pegasus [17], processes a single big graph like social network or web data instead of set of small graphs.

3 PRELIMINARIES

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 $\subseteq$V x V. L is the set of labels of vertices or edges, and the labelling function defines the mapping: V $\cup$ E $\rightarrow$ L. We also denote the vertex set and the edge set of graph g by V(g) and E(g) respectively.

Subgraph Isomorphism Given two graphs s = (Vs, Es, Ls, ls) and g = (Vg, Eg, Lg, lg), s is said to be subgraph 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)) $\in$ Vg, and fs[u, v] = fg[f(u), f(v)]. Given two graphs g = (V, E, L, l) and g1 = (V 1, E1, L1, l1), g is sub-graph isomorphic to g1, denoted g $\subseteq$ g1 if and only if there exists an injective function. The function is injective (one-to-one) if every element of the codomain is mapped to by at most one element of the

domain. An injective function is an injection. ∀x,y∈A f(x)=f(y)⇒x=y or equivalent ∀x,y∈A,x=y⇒ f(x) = f(y).

3.2 Problem Statement

For the given graph dataset D = g1, g2, g3 gn contains set of small graphs and query set Q = q1, q2, q3..qm contains set of query graphs such that | Q | | D | for each graph query q ∈ Q, we have to find all the graphs to which q is subgraph isomorphic from D.

Streams of queries are processed at a time in this approach. In a distributed environment, we have to process queries that emerge from different sources. Efficient query processing on high speed stream is useful for many applications that require accurate query response. Higher throughput is achieved by eliminating the recurring common parts among query in batch query processing Technique.

3.3 Graph Representation

Graph Representation as One Single Line:

Number of Graph datasets are available in multi-line format. Frequent subgraph mining works[2] divides the given input graph dataset into a set of files and are given as input to the Map Reduce program as part of the data representation step. A serialized format of g, exists in a single line format, which enumerates vertices and edges in g, i.e {|V(g)|, |E(g)|, l(V(g)), E(g)} where e ∈ 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 vertces: total no of vertices in the graph. No of edges: total no 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.

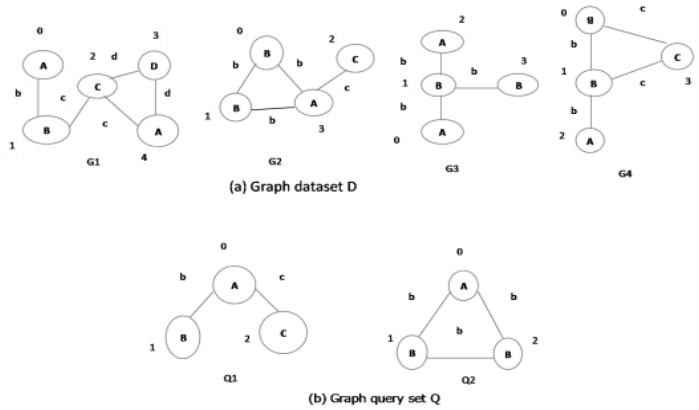


Figure 1: A set of data graphs and a set of query graphs

4 PROPOSED APPROACHES

In this section, we first discuss a naive approach for processing multiple graph queries using MRDGG.

4.1 Naive Approach

Naive approach performs parallel subgraph isomorphism test for each pair of query graph q and data graph g such that q ∈ Q and g ∈ D. Since | Q | | D |, using Distributed Cache these query graphs are available to all machine in the cluster. 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. subgraph isomorphism test takes | Q | x | D | times in total, which is a time consuming process for query processing. To overcome this we proposed the MapReduce based Distributed Graph Grep (MRDGG).

4.2 MRDGG: MapReduce based Distributed Graph Grep

MRDGG is processing multiple graph queries using Replicated join using Distributed Cache and the number of subgraph isomorphism tests are reduced by filter and verify scheme. The basic steps of Distributed Graph Grep are as follows:

• Build the Inverted Edge Occurrence based Index for large graph database.

• Search space is going to be reduced by filtering the database on the basis of submitted queries.

• Perform exact matching.

• Index maintenance.

Proposed work is divided into three phases:

Index Building, Multiple Sub-graph query processing and Index maintenance phase. The Figure 2 below describes in detail about each phase. According to the filter and verify approaches, during filter step we acquire candidate data graphs then in the second step instead of performing the subgraph isomorphism tests on all data items apply only on the candidate graphs. Filter step filters all irrelevant graphs by comparing the features of query graphs with the features of data graph. It reduces the overall execution time significantly by reducing the number of graphs to be verified.

It is using index files to filter and verify. We design three approaches to graph query processing using filter and verify scheme.

We design two Index files which are useful for Graph Query Processing. Inverted Edge Index file is used to filter the data graphs. Inverted Edge Occurrence Index is used to verify the subgraph isomorphism testing. Figure 2 shows that one map reduce round is used for index creation and one round is used for index maintenance.

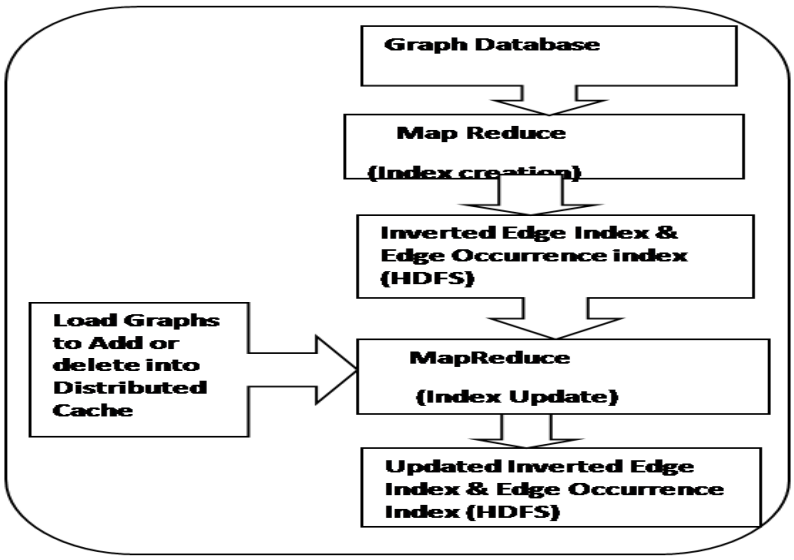


Figure 2: An overview of Index Building and Index Maintenance

According to the Index files used in Graph Query processing we propose two approaches as follows:

• Distributed Graph Query Processing using Inverted Edge Occurrence

Index (One MapReduce round)

• Distributed Graph Query Processing using both Inverted Edge Index and Inverted Edge Occurrence Index (two MapReduce rounds)

Mapper Output

Edge	Graph Set
<A B b>	G1(0,1)
<B C c>	G1(1,2)
<A C c>	G1(4,2)
<A B b>	G2(3,0),(3,1)
<A C c>	G2(3,2)
<B B b>	G2(0,1)
<C D d>	G1(2,3)
<A D d>	G1(4,3)
<A B b>	G3(2,1),(0,1)
<B B b>	G3(1,3)
<A B b>	G4(2,1)
<B B b>	G4(0,1)
<B C c>	G4(1,3)(0,3)

Reducer Output

Edge	Graph id with Edge Occurrences
<A B b>	G1(0,1),G2(3,0)(3,1),G3(2,1)(0,1),G4(2,1)
<B B b>	G2(0,1),G3(1,3),G4(0,1)
<A C c >	G1(4,2),G2(3,2)
<B C c>	G2(1,2),G4(1,3)(0,3)
<C D d>	G1(2,3)
<A D d>	G1(4,3)

Figure 3: Index building Phase

4.3 Distributed Graph Query Processing using Inverted Edge Occurrence Index

Inverted Edge Occurrence Index: With the help of graph database file map reduce job builds inverted edge occurrence index.

• Parse the graphs: Divide the graph into edges.

• Find each edge and its graph ids and occurrences list: Each mapper sends edge and its graph id and edge vertex ids these vertex ids are called occurrence. Here we are using combiner to get local Graph ids and occurrence list 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 identical edges are aggregated together in the reduce function

<Edge,GraphNo(Occurrence list)> pairs with identical Edge are aggregated together, and the

reducer function generates the out- put file Edge index and other one is Edge Occurrence List Index.

Algorithm1:InvertedEdgeOccurrenceIndex

Result:EdgeOccurrenceIndex

Data:InputgraphdatabaseD

1ClassMapper

2methodmap(Key:Offset,Value:Gcode(gid,graph));

3foreachgraph$G_i$ingraphdatabaseDdo

4foreachedgelabeledwithEdgeLabelido

5Output($EdgeLabel_i$,$G_i$andOL_i)

6end

7end

8ClassReducer

9methodreduce(Key:Edge,Value:List(occurrenceslist))

foreachedge labeledwithEdgeLabeldo

10 Concatenateallgraphidstogenerate GraphSetstring GS_i:G_{i1} andOL_{i1},G_{i2}andOL_{i2},G_{i3}andOL_{i3}...G_{in}andOL_{in}

emit($EdgeLabel_i$,List ofGS_iandOL_i)

11end

Inverted Edge Occurrence Index File Format is as follows:

Edge - List of graph ids and its occurrences

A B c - g1 (1,2)(3,5) , g2 (1,3)

This assigning index is presented in Algorithm 1.

This approach is using one map reduce round for filter and verification. In this approach Inverted Edge Occurrence Index is the input instead of original graph Database for verification because the edge occurrence positions are there in the index. The inverted Edge occurrence index is read from HDFS and query graph edge occurrence index is loaded into distributed cache. Mapper is reading an Edge from Inverted Edge occurrence list and verifying if it exists in the query edges. If it exists, Mapper is doing join step and send <query id and data graph id> as the key and <Edge Lable and edgeoccurrences> as the value. Reducer get query id and graph id as key and all its edges and edge occurrences as values. Reducer is doing verification. In this approach we can get false positives.

Query Processing using Inverted Edge Occurrence Index

This approach is using single MapReduce to do filter and verification. Query preparation step is same as above approach. This approach is using inverted edge occurrence index to join the query edges and data graph edges. Here inverted edge occurrence index is input to the mappers. Each mapper is doing join step based on edges and send it to the reducer. Reducer is doing filter step and verification. Query preparation phase: Query preparation Step: Set of query graphs which we are going to process are preprocessed in this step. We prepare Inverted edge Occurrence index including occurrences list for both filter step and verification step, and is loaded via Distributed Cache. In this round the inverted edge occurrence index file is the input. Query edge index file is loaded into distributed cache. At the map stage the input data are partitioned into equal size blocks and each block is assigned a mapper. Each mapper reads block of inverted edge occurrence index. From the input each edge and its graph ids and occurrences are coming, if that edge is there in the query index then do join operation and prepare the key and value and send to the reducer.

Reducer: At the reducer, key is query id and data graph id and value is he list of edge and its occurrences in that graph. Each line of reducer is one data graph id and corresponding edges. Now get the edges from query, if all the edges in the query are present in the data graph then do verification otherwise filter the graph. Like this reducer is doing both filter and verification.

4.3.1 Distributed Graph Query Processing using both inverted edge index and Inverted Edge occurrences index

This approach requires two MapReduce rounds. One round is used for filter and one round is used for verification. This approach uses both index files and it gives the result.

Filter step is same as in the first approach Algorithm 2. Verification: During filter step, Inverted Edge index is used

---

Algorithm2:Filter and verification using EdgeOccurrenceIndex

Result:Finalresult:queryid-graphids

Data:InvertedEdgeOccurrenceIndex,Indexofquerygraphsare loadedintoDistributedCache

1methodSetup

2readindexofquerygraphsfromintoqueryedgelistfromdistributed cache

3classMapper

4methodmap(key:offset,value:edge-gid(occurrencelist))

5getedgefrominputfile

6searchinqueryedgelist

7ifedgefoundinqueryedgelistthen

8getgraphidslistintogidsandoccurrences;

9getqueryidslistintoqids;

10foreachqidinqidsdo

11key=qid+gid;

12value=edge+occurrencelist;

13emit(key,value);

14end

15classReducer

16methodReduce (key:qid-gid,value:edgeoccurrencelist)

17getqueryidanditsinformation;

18ifsubgraphisotest(qid,gid)then

19emit(qid,gid);

and inverted edge index of query graphs is placed into distributed cache. This filter step is giving the candidate graphs which contain all the edges of the query graph (it is same as filter in first approach). So False positives are reduced. During verification, Inverted Edge Occurrences Index is the in- put and query edge occurrences are placed in distributed cache. In this round mapper is doing join step to get the edge occurrences list of the candidate datagraphs only. It is giving work to all the datanodes. It reduces communication cost. Even though it is using two MapReduce rounds, it is reducing the communication cost because it is sendingonly the edges in query graph instead of all the edges in the data graph. The process is shown in figure 5.

AIgorithm3:VerificationusingEdgeOccurrenceIndex

Result:Finalresult:queryid-graphids

Data: InvertedEOIndexfromHDFA,filterstepresultandquery graphsfromDistributedCache

1methodSetup

2readinvertedeoindexofquerygraphsintoqueryedgelist;

3readfilterstepresultintofilterquery;

4classMapper

5methodmap(key:offset,value:edge:gid(occurrencelist))

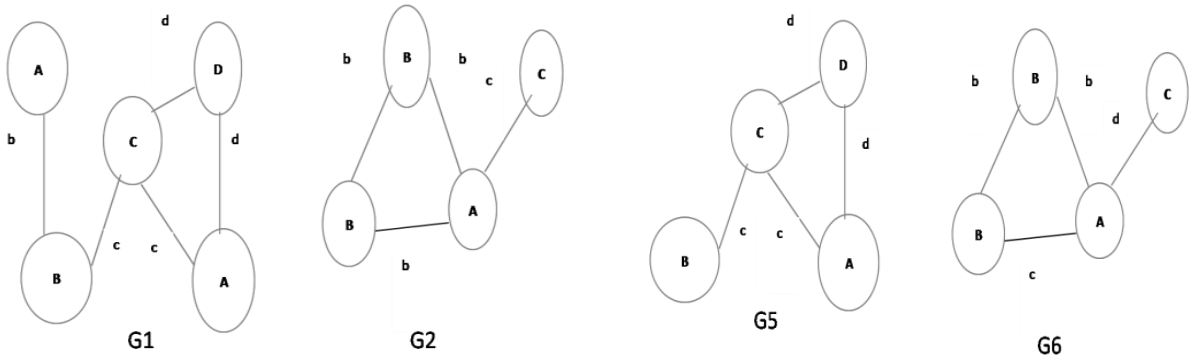
6getedgefromvalue

```
7searchinqueryedgelist
8ifedgefoundinqueryedgelistthen
9getgraphidslistintogidsandoccurrences
10getqueryidslist intoqids
11foreachqidinqidsdo
12getresultgidsfromfilterquery(qid;
13foreachgidinresultgids;
14occurrencelist=resultgids.get(gid);
15key=qid+gi;
16value=edge+occurrencelist
17emit(key,value)
18end
19ClassReducer
20methodreduce(key:qid-gid,Value:edgeoccurrencelist)
21getqid-gidanditsedgeoccurrencelist
22getqueryidanditsinformation
23ifsubgraphisotest(qid,gid)then
24output(qid,gid)
```

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 new graphs are added to the index and delete graphs are deleted from the index using single MapReduce round.

All the graphs which we want to add or delete are prepared as update inverted edge index as pre-processing step. 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).



(a) Graphs to delete (b) Graphs to insert

Figure 4: Graphs to insert into/delete from the Graph Database

4.4.1 Incremental maintenance of inverted Edge Index and Inverted Edge Occurrence Index

This is the third step of MRDGG. In this step we are updating the index files. We have to do the two steps adding or removing index in graphs is also termed as index maintenance.

• Graphs are added to the inverted index

• The graphs are deleted from inverted index

• The data graph to which we want to assign index must be loaded into

Distributed cache

• The data graphs which we want to delete from index should be loaded into distributed cache

• New graphs are assigned inverted edge index

• Assign the inverted edge index for the graphs that are going to be deleted

• Join operation is performed based on the edge with add graph list

• Join operation is performed based on the edge with add graph list

• When new graph id's are added then the inverted index is updated.

• After deleting the group id's update the inverted index

• Output the updated list to updated index file

This type of indexing takes less time to read a particular file which contains the edge. Simultaneously all machines are involving in the updating process.

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 to delete and insert graphs.

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

Figure 5: Edge index

Number of Graphs	NumberofGraphs	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

Table 1: Real Life Biological Datasets

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.

6 EXPERIMENTS

Experimental results are briefly described in this section. The performance of MRDGG on synthetic and real datasets was analyzed. The two indexing implementations in MapReduce are compared in our experiment described the existing implementations outcomes later the proposed method outcomes are described in detail.

6.1 Experimental setup

6.1.1 Datasets

The experiments were conducted using the dataset shown in Table 1, these collected from online source PubChem website. PubChem has more than one million chemical structures. Each graph contains 24.98 vertices, 26.76 edges, 4.5 distinct vertex labels, 3.0 distinct edge labels on average, and the total number of distinct vertex labels and distinct edge labels is 82 and 4, respectively. We

generated the synthetic dataset of size 3 lakh graphs and graph queries, which contain various number of queries ie | Q | = 100, 200, 300 upto1000.

6.1.2 Implementation platform

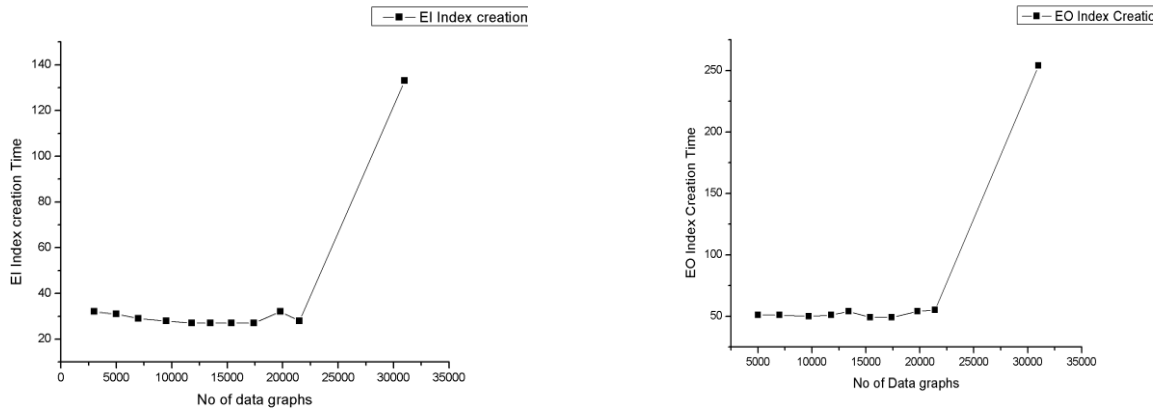
The implementation platform is java and Hadoop (version 1.2.1) an open source version of MapReduce. The data is stored in Hadoop Distributed File System(HDFS) an open source implementation of Google File System GFS[18]. A local cluster with 8 nodes is used to test our experiment. Front-end: HP Proliant DL380P Gen8

2 x Intel Rxeon R 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 Node Intel R xeon R CPU E5-264 0 (2.5 GHz / 6-core/15MB / 95w) Processor, 16GB RAM, 2* 300GB HDD Intel 7500 chip set with node controller.

6.2 Experimental results

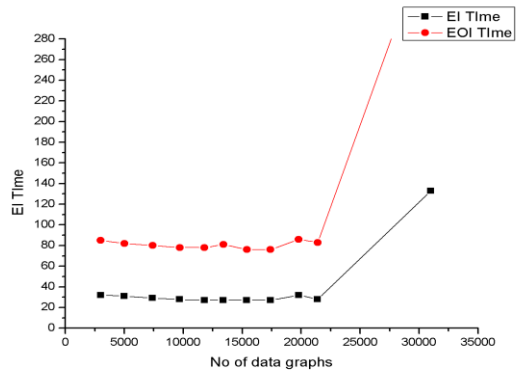
6.2.1 Index Building Time versus Number of Data Graphs

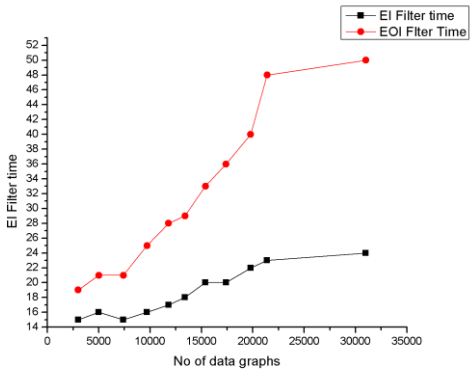
Multiple MapReduce indexing strategies efficiency is determined through this experiment. Figure 8 shows how the time taken by different indexing strategies. We observe that the edge indexing is taking less time compared to edge-occurrence indexing. Edge occurrence indexing has to transfer vertex ids this increase the communication cost also compared to edge indexing. Figure 12 (b) shows the time taken for index creation for real data sets showed in Table 1. After adding edge count to the indexing strategies then it is taking more time. As number of graphs is increasing time is taking more.



a) Inverted Edge Index Building Time (b) Inverted Edge OccurrenceIndex Building Time

Figure 8: Index building Time for both indices



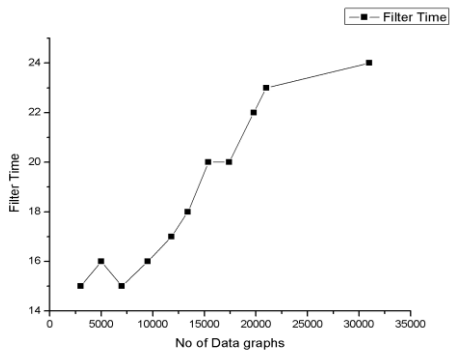


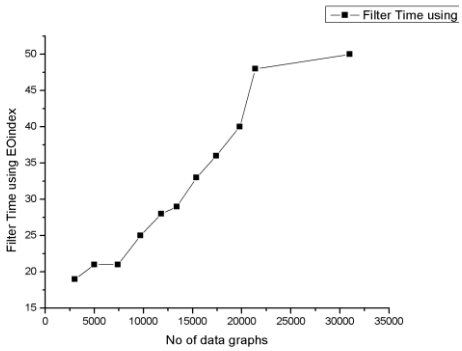
(a) Comparision of Both Indexes (b) Inverted Edge Index Time for Real Datasets

Figure 9: Index building Time comparision

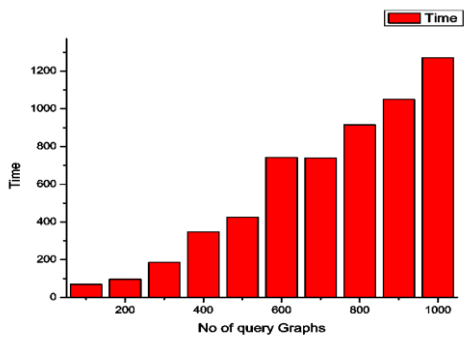
6.2.2 Filter time for different number of data graphs

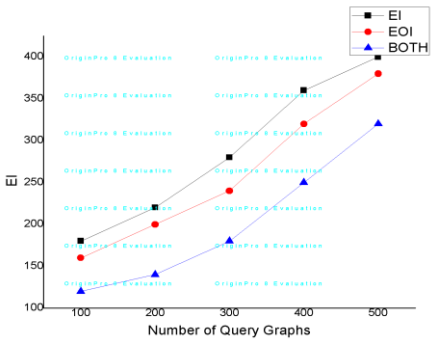
To find how much time is taking for query processing using different indexing strategies and for different number of data graphs





(a) Filter time using Inverted Edge Index (b) Filter time using Inverted Edge Occurrence Index





(c) Comparison of Filter time using both (d) Comparison of Filter time using count indexes

Figure 10: Filter Time for Different Approaches

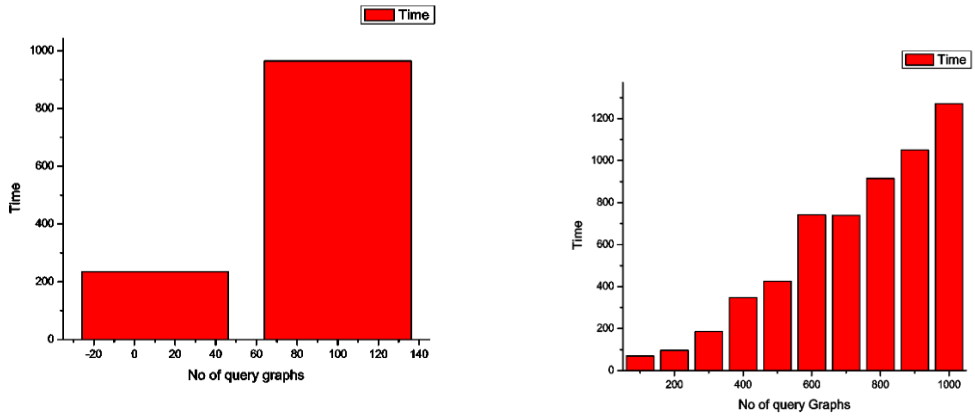
. Figure 10 shows that Edge Indexing is taking less time for filtering and is not used for verification, so we have to depend on the original graph database. So it is taking more time. Edge Occurrence Index is taking some more time compared to EI, but with only one round of mapreduce is sufficient for both filter and verification.

6.2.3 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. Sometimes if query graphs are not matched with data graphs then it takes less time because in the filter itself we eliminated false positives.

6.2.4 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 12 shows the comparison.



(a) Query Processing Time for synthetic data sets (b) Query Processing Time for Yeast data sets

Figure 11: Query Processing Time vs Number of Query Graphs

6.2. 5 Comparison of EI and EOI and Both Approaches

In this experiment we conducted experiments using all three approaches. EI is taking less time compared to the EOI. Because EI is doing filter efficiently. EOI is taking more time for both filter and verification steps. If we combine both the approaches it is taking less time and less communication cost.

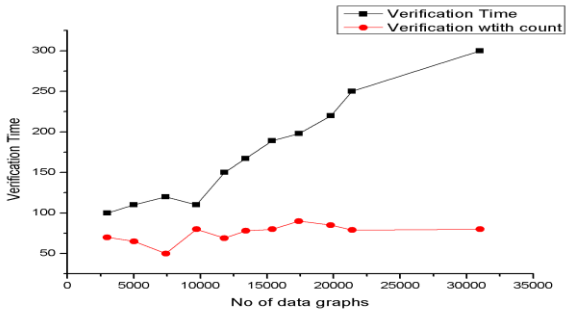


Figure 12: Verification Time

Figure 13 shows the comparison of three techniques. From that we can understand third technique is the best.

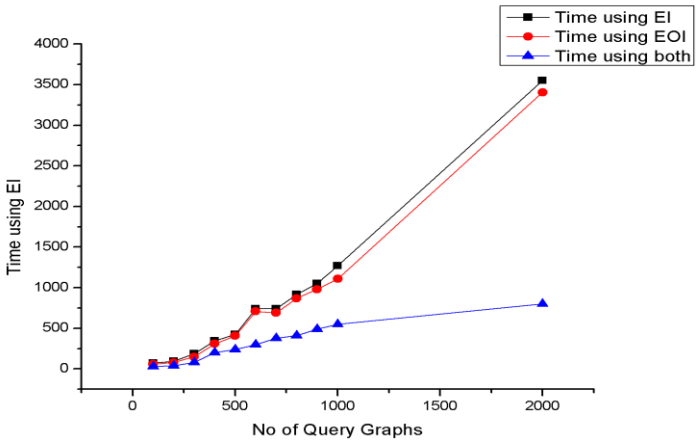


Figure 13: Comparison of three Approaches

7. CONCLUSIONS

In this paper we present multiple graph query processing using Join Technique in MapReduce. Two indexing methods are proposed and used for batch graph query processing using MapReduce. We show the performance of MRDGG over real life and large synthetic datasets for various number of query graphs. We also compare the filtering rate after optimization. In future work, we will study the use of graph index to mine frequent subgraphs from a large graph database.

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 Iterative 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.243–271, 2005

[4] Giugno, R., Shasha, D: Graph Grep: A Fast and Universal Method for Querying Graphs. Proceedings of ICPR 2, 112-115 (2002)

[5] Yan,X.,Yu,P., Han, J.: Graph Indexing Based on Discriminative Frequent Structure Analysis. ACM Transactions on Database Systems 30(4),960-993(2005)

[6] Cheng, J., Ke, Y., Ng, W., Lu,.: FG-Index: towards verification-free query processing on graph databases in: Proceedings of ICDE (2007)

[7] He,H.,Singh, A.K.:Closure-Tree.: An Index Structure for Graph Queries. In Graphs, Proceedings of ICDE (2006)

[8] Haoliang Jiang ; Haixun Wang ; Yu, P.S. Shuigeng Zhou,: GString: A Novel Approach for Efficient Search in Graph Databases Proceedings of ICDE (2007)

[9] Song-Hyon Kim, Kyong-Ha Lee, Hyebong Choi and Yoon-Joon Lee, "Parallel Processing of Multiple Graph Queries Using MapReduce",DBKDA,2013

[10] Fathimabi Shaik, R.B.V. Subramanyam, and D.V.L.N. Somaya- julu. ”MSP: Multiple Sub-graph Query Processing using Structure-based Graph Partitioning Strategy and Map-Reduce.” Journal of King Saud University-Computer and Information Sciences (2016).

[11] Fathimabishaik, R.B.V. Subramanyam, D.V.L.N Somayajulu,”Map- Reduce based Multiple SubGraph Enumeration Using Dominating-Set Graph Partition”, International Journal of Information Engineering and Electronic Business(IJIEEB), Vol.9, No.2, pp.36-44, 2017. DOI:10.5815/ijieeb.2017.02.05.

[12] Willet, P.: Chemical similarity searching. J.Chem. Info. Comput.Sci.
38,983-996(1998)

[13] Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large cluster. In: Proceedings of OSDI, pp 137-150(2004)

[14] James Cheng, YipingKe, Ada Wai-chee Fu, Jeffrey Xu Yu: Fast Graph Query Processing with a Low-Cost Index.VLDB 2011.

[15] YifengLuo,Jihng Gun ndShugeng Zhou, Towards Efficient Subgraph- Search in Cloud Computing Environments. Spinger-verlag Berlin Heidel- berg 2011.

[16] 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.135–146.

[17] U. Kang, C. Tsourakakis, and C. Faloutsos, “Pegasus: mining peta- scale graphs,” Knowledge and information systems, vol. 27, no. 2, pp.303–325, 2011.

[18] S. Ghemawat, H.Gobioff, S.T.Leung. The Google File System, in:proceedings of the 19th ACM Symposium on Operating Systems Prin- ciples, vol.37 of SOSP ’03, ACM, New York, USA, 2003.

[19] S. Blanas, J. Patel, V. Ercegovac, J. Rao, E.Shekita, and Y. Tian, “A comparison of join algorithms for log processing in mapreduce,” in Proceedings of the 2010 international conference on Management of data. ACM, 2010, pp. 975–986.

[20] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In ICDE, pages 62–73, 2013

[21] Yarramalli, Sai Sravya, et al. "Digital Procurement on Systems Applications and Products (SAP) Cloud Solutions." 2020 Second International Conference on Inventive Research in Computing Applications (ICIRCA). IEEE, 2020.

[22] S. Fathimabi, E. Jangam and A. Srisaila, "MapReduce based Heart Disease Prediction System," 2021 8th International Conference on Computing for Sustainable Global Development (INDIACom), 2021, pp. 281-286, doi: 10.1109/INDIACom51348.20