Finding Pattern in Large Graph Database using Mapreduce (Hadoop)

M. Rammurthy
Department of Information Technology
Anna University Regional Campus Coimbatore, India

Dr. P. Marikkannu
Department of Information Technology
Anna University Regional Campus Coimbatore, India

Abstract

Handling the massive data in a single database is more complex to process and mining. So, Data is stored in Distributed manner and it co-ordinates together by mapreduce framework. In real world prediction plays a vital role. The same way, Here in biological dataset pattern mining is possible through finding relationship between them. Finding relationship and co-relating the words (nodes) is quite simple in graph. So, we used graph structured database. An important node pair is referred as candidate key pair to match the entire frequent subgraph in particular partitioned data. An iterative mapreduce is used to generate global frequent subgraph mining in distributed database.

Keywords: Frequent subgraph Mining, candidate pattern generation, Graph Data, Iterative Mapreduce, Distributed database

I. INTRODUCTION

In traditional frequent subgraph mining algorithm assumed that data is fit to the main-memory for processing but in later data growth rate is tremendously increased so, we can't execute in singe database. After facing this problem they decided to implement the parallel process execution in same function itself even though it's perfection is high it increase the burden to programmers because they need to concentrate on both logic and parallelization technique to overcome this complexity a java supported framework is developed to take care of parallelization technique. so, programmers can concentrate on their logics. And now we were going to find pattern in biological DNA sequence data in distributed database by the support of mapreduce technique. MapReduce is a simple programming model for processing huge data sets in parallel manner. The basic notion of MapReduce is to divide a task into subtasks, handle the sub-tasks in parallel, and aggregate the results of the subtasks to form the final output. Programs written in MapReduce are automatically parallelized: programmers do not need to be concerned about the implementation details of parallel processing. Instead, programmers write two functions: map and reduce. The map phase reads the input (in parallel) and distributes the data to the reducers. Auxiliary phases such as sorting, partitioning and combining values can also take place between the map and reduce phases. MapReduce programs are generally used to process large files. The input and output for the map and reduce functions are expressed in the form of key-value pairs.

II. LITERATURE REVIEW

Jeffrey Dean and Sanjay Ghemawa uses the non-iterative method to find the frequent subgraph mining. And those map function and reducer function always gives key and value pairs. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

S. Hill, B. Srichandan, and R. Sunderraman implemented iterative method to find frequent subgraph but due to the duplication of subgraph basic isomorphic technique is used to find similar graph. Based on normal classification dependency graph is separated so accuracy of pattern is decreased. So, Markova chain clustering method is used to overcome the dependency problem. One of the challenging problems in graph mining is about the completeness because the of complexity graph structures.

III. PROPOSED MODEL

FSM method computes the support of a candidate subgraph pattern over the entire set of input graphs in a graph dataset. In a distributed platform, if the input graphs are partitioned over various worker nodes, the local support of a subgraph in the respective partition at a worker node is not much useful for deciding whether the given subgraph is frequent or not. Also, local support of a subgraph in various nodes cannot be aggregated in a global data structure, because, MapReduce programming model does not provide any built-in mechanism for communicating with a global state. So, iterative mapreduce is used for global communication.

A. *System Architecture:*

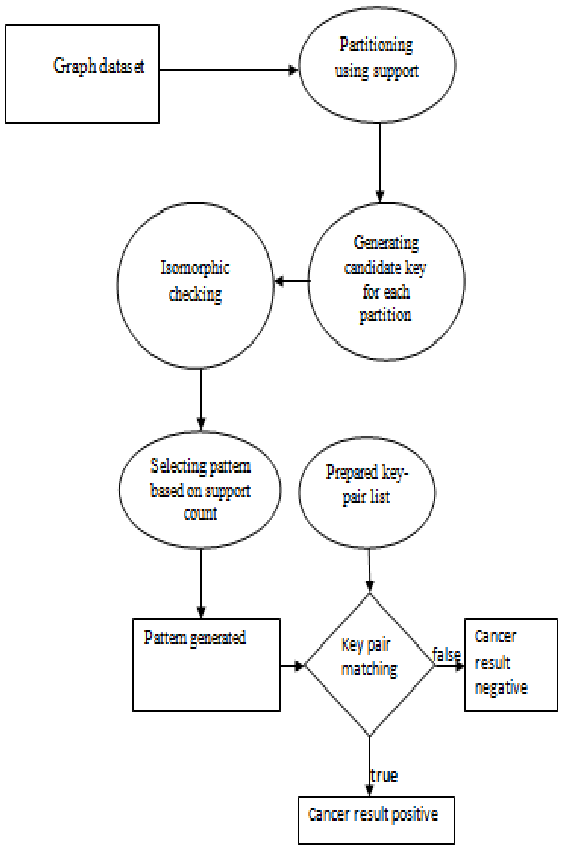

Fig. 1: System architecture

The figure 1 shows the system architecture diagram of the proposed system. The input dataset is partitioned using support count by pearl language configuration method, candidate pattern generation is done by canonical coding scheme. After candidate pattern generation duplication can be reduced by isomorphic checking, based on the support value count pattern is generated. finally we need to match both generated pattern and prepared key pair list .If it matches cancer result positive or negative.

IV. Implementation and Results

Following modules are:
- Partitioning data
- Generating candidate pattern
- Isomorphism checking
- Support count
- Mining frequent subgraph patterns

A. *Partitioning Data:*

Data can be partition based on the support value (i.e) no.of edges in the partition. And each transactions of the nodes is limited based on the support value and we can partition the data based on the data schema partitioning can be done through pearl configuration functions.

1) *Pseudocode:*

```
BEGIN
   Partition(D)
            Create a data directory in hdfs
            Upload document d
            WHILE data present in D
             Partition=create.partition()
Write partition to file partition i       in data directory
            END WHILE
END
```

B. Generating Candidate Pattern:

If K is a size of the present node then we will adjoins the next adjacent nodes to parent node. Additional vertex of a forward edge is given an integer id, which is the largest integer id following the ids of the existing vertices of c, thus the vertex-id stands for the order in which the forward edges are adjoined while building a candidate pattern. Note that, if d has k þ 1 edges, based on the order how its edges have been adjoined, d could have many different generation paths in a candidate generation tree; however, in all FSM algorithms, only one of the generation paths is considered valid, so that multiple copies of a candidate pattern are not generated.Simple put, “right most vertex” (RMV) is the vertex with the largest id in a candidate subgraph and “right most path” (RMP) is the shortest path from the lowest id vertex to the RMV strictly following forward edges.

1) Map Function:

```
BEGIN
Mapper(Long key ,ByteWritable value)
   P=reconstruct_pattern(value)
          s=candidate_generationon(p)
           FOR all p(i) in p
            IFisomorphic_test(p(i))=true
               IF length(p(i).oL)>o
                                 Intermediate_key=min_dfs_code(p(i))
                                          Intermediate_value=serialize(p(i))
                                          Emit(intermediate_key,intermediate_value)
ENDIF
ENDIF
ENDFOR
END
```

2) Redue Function:

```
//key=min_dfs_code
//values=List of bytes.streamof a pattern objects in all partitions
BEGIN
Reducer(Text  k ey, BytesWritable values)
   FOR all value in value
         Support+=get_support(values)
   ENDFOR
Emit(long support)
END
```

C. Isomorphism Checking:

A candidate pattern can be generated from multiple generation paths, but only one such path is explored during the candidate generation step and the remaining paths are identified and subsequently ignored. To identify invalid candidate generation paths, a graph mining algorithm needs to solve the graph isomorphism task, as the duplicate copies of a candidate patterns are isomorphic to each other. A well-known method for identifying graph isomorphism is to use canonical coding scheme, which serializes the edges of a graph using a prescribed order and generates a string such that all isomorphic graphs will generate the same string. There are many different canonical coding schemes; min-dfs-code is one technique. According to this scheme, the generation path of a pattern in which the insertion order of the edges matches with the edge ordering in the min-dfscode is considered as the valid generation path, and the remaining generation paths are considered as duplicate and hence ignored. FSM-H uses min-dfs-code based canonical coding for isomorphism checking.

```
BEGIN
Isomorphism_test(P)
   FOR all p(i) in P
      Scanning all node names of a graph in p (i)
                    Sorting names in alphabetic order
      Store the duplication value
      Emit(duplication value)
   ENDFOR
END
```

D. Support Counting:

Support counting of a graph pattern g is important to determine whether g is frequent or not. To count g’s support we need to find the database graphs in which g is embedded. This mechanism requires solving a subgraph isomorphism problem, which is NP complete. One possible way to compute the support of a pattern without explicitly performing the subgraph isomorphism test across all database graphs is to maintain the occurrence-list (OL) of a pattern; such a list stores the embedding of the pattern (in terms of vertex id) in each of the database graphs where the pattern exists. When a pattern is extended to obtain a child pattern in the candidate generation step, the embedding of the child pattern must include the embedding of the parent pattern, thus the occurrence-list of the child pattern can be generated efficiently from the occurrence list of its parent. Then the support of a child pattern can be obtained trivially from its occurrence-list.

```
BEGIN
Supportcount(int support)
IF support>=minimum_support
   FOR all values invalues
      Write_in_file(key,value)
   ENDFOR
ENDIF
END
```

E. Mining Frequent Subgraph Patterns:

Based on the support count occurrence list is prepared to increase the efficiency of generated pattern. If the support is higher than the minimum support threshold (minsup), the given pattern is frequent, and is stored in the separate set. And generated pattern is uploaded in HDFS.

Table - 1
Runtime of FSM-H in Biological DNA Sequence Data

SIZE OF GRAPH	*TOTAL RUN TIME (IN MIN)*	*MAPPER TIME*	*REDUCER TIME*
100k	*25.4*	*6.54*	*18.86*
150k	*38.1*	*9.81*	*28.29*
200k	*50.8*	*13.08*	*37.72*

When comparing both mappers phase and reducer phase. Reducer phase will take maximum amount of time because it must perform both sorting and shuffling. At the same time reducer phase must wait for some time.

V. CONCLUSION

A novel iterative MapReduce based frequent subgraph mining algorithm, called FSMH. We show the performance of FSM-H over real life and large synthetic datasets for various system and input configurations. We also compare the execution time of FSM-H with an existing method, which shows that FSMH is significantly better than the existing method. One possible solution to improve the communication complexity of present system is to use distributed cache for storing the static data structures so that they do not require to be piggybacked with the pattern objects over the networks.

REFERENCES

[1] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on larg clusters,” Commun. ACM, vol. 51, pp. 107–113, 2008.
[2] S. Hill, B. Srichandan, R. Sunderraman, “An iterative Mapreduce approach to frequent subgraph mining in biological datasets,” in Proc. ACM Conf. Bioinformat., Comput. Biol. Biomed., 2012 pp. 661–666.
[3] S. N. Nguyen, M. E. Orlowska, and X. Li, “Graph mining based on a partitioning approach,” in Proc. 19th Australasian Database Conf., 2008, pp. 31–37.
[4] F. Afrati, D. Fotakis, and J. Ullman, “Enumerating subgraph instances using map-reduce,” in Proc. IEEE 29th Int. Conf. Data Eng., Apr. 2013, pp. 62–73.
[5] M. Kuramochi and G. Karypis, “Frequent subgraph discovery,” in Proc. Int. Conf. Data Mining, 2001, pp. 313–320.
[6] J. Huan, W. Wang, and J. Prins, “Efficient mining of frequent subgraphs in the presence of isomorphism,” in Proc. 3rd IEEE Int. Conf. Data Mining, 2003, pp. 549–552.
[7] G.-P. Chen, Y.-B. Yang, and Y. Zhang, “Mapreduce-based balanced mining for closed frequent itemset,” in Proc. IEEE 19th Int. Conf. Web Serv., 2012, pp. 652–653.
[8] X. Xiao, W. Lin, and G. Ghinita, “Large-scale frequent subgraph mining in Mapreduce,” in Proc. Int. Conf. Data Eng., 2014, pp. 844– 855.