A STUDY OF GRAPH PARTITIONING TECHNIQUES FOR FAST INDEXING AND QUERY PROCESSING OF A LARGE RDF GRAPH

A THESIS IN

Computer Science

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

By

DINESH BARENKALA

B.E, Osmania University, 2009

Kansas City, Missouri
2013

A STUDY OF GRAPH PARTITIONING TECHNIQUES FOR FAST INDEXING AND QUERY PROCESSING OF A LARGE RDF GRAPH

Dinesh Barenkala, Candidate for the Master of Science Degree

University of Missouri-Kansas City, 2013

ABSTRACT

In recent years, the Resource Description Framework (RDF) [34] has become increasingly important for the Web and in domains such as defense and healthcare. Companies such as the New York Times [40], Best Buy [39], and Pfizer are leveraging RDF and other Semantic Web technologies for data management. Using RDF, any assertion can be represented as a (subject, predicate, object) triple. The collection of triples together represents a graph. Many techniques have been developed for RDF indexing and query processing and the most popular among them store and process RDF data using an RDBMS.

In this thesis, we study the impact of existing graph partitioning techniques on indexing and query processing of a large RDF graph (e.g., YAGO [40]) with millions of edges and vertices. Our goal is to partition a large RDF graph into smaller graphs and then index the smaller graphs efficiently for faster query processing. In order to cope with cut edges, we compute the 2-hop distance across each cut edge. Once partitions are computed, we construct an index using a recently developed technique called RIS. Queries are also processed using RIS. We report the benefits and trade-offs of two different partitioning strategies using the YAGO dataset on metrics such as index construction time, index size, and query processing time. The first partitioning strategy treats the original RDF graph as an

unweighted graph during partitioning. The second strategy treats the original graph as a weighted graph during partitioning. We compared the results obtained by RIS (on partitioned graphs) with RDF-3X [38], a state-of-the-art RDF query processing engine.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a thesis titled "A Study of Graph Partitioning Techniques for efficient RDF Query Processing", presented by Dinesh Barenkala, candidate for the Master of Science degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Praveen R. Rao, Ph.D., Committee Chair

Department of Computer Science Electrical Engineering

Yugyung Lee, Ph.D

Department of Computer Science Electrical Engineering

Lein Harn, Ph.D

Department of Computer Science Electrical Engineering

TABLE OF CONTENTS

ILLUSTRATIONS

TABLES

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Dr. Praveen Rao for giving me an opportunity to be part of his research team. His support and encouragement motivated me to pursue research and successfully complete my Master's Thesis. Dr. Rao also supported me financially that helped me dedicate all of my time to research. This work was supported in part by the National Science Foundation under Grant No. 1115871 and a grant from University of Missouri Research Board. I would like to thank Vasil Slavov for helping me with optimizing my code and Srivenu Paturi for helping me with RIS indexing. I would like to thank Priyanka, my fiancée for her support throughout my thesis. Finally, I want to thank my family for their love and support throughout these years.

CHAPTER 1

INTRODUCTION

In the agile world, the augmentation of web technologies is very rapid that information is beyond manageable and here comes the RDF(Resource-Description Framework) [34] to access enormous data in the form of compact graphs. RDF data model is increasing its importance day-by-day in the World Wide Web and other domains such as health care and defense. It is more suited than relational data model for storing certain types of knowledge base. Big Companies such as Best Buy [35], New York Times [40], BBC, Pfizer, etc., are looking towards RDF and other semantic web technologies for data management.

Today, there are a number of open-source and commercial tools for storing and querying RDF graphs (e.g., 3Store [24], Big Data, Jena [19], etc). These tools either use traditional RDBMS or a native RDF database, for storing and processing RDF. Processing, indexing and querying a huge RDF graph on a single machine is a big challenge in terms of the resources and time. The incredible fact that it takes few days to get the results from a query paved a way to search for effective ways of querying. This lead to use of parallel machines, cluster of nodes, many core processors, etc.

Currently, there are many very large RDF datasets available to everyone (YAGO2 [14], LUBM [31], Uniprot RDF [17], DBPedia [37], etc). It is known fact that the RDF data can be represented as a graph. If this large graph can be partitioned into numerous small graphs using any of the graph-partitioning algorithms and the queries are run on these trivial graphs, it would be a breakthrough for querying. Considering the fact that queries on these

trivial graphs run faster on a single machine (parallel machines have overhead of collecting and combining the results), we can compute the results more effectively and quickly.

We plan to use RIS (RDF Indexing via Signatures), new approach that outperforms RDF-3X for very large queries (more number of joins), for indexing and querying the graphs that we get after graph partitioning. We also have plots that compare the query time of RIS and RDF-3X.

1.1 Focus of this Thesis

The idea of Graph partitioning can expedite the process of querying huge graphs. But, the overhead with Graph partitioning is that we may lose some of the results due to the cut edge. Any edge that is cut in order to partition a graph into smaller graphs can be called as cut-edge. It is like a bridge between the two partitions which no longer exists after partitioning. It does not belong to any of the partitions. So, when we have a big query that is used to query this partitioned graph (many small graphs), it is most likely that our query spans across the cut-edge, thus we may lose some of the results. So, handling the cut edges is very important for an effective graph partitioning. In this thesis we propose a design on how we handle the cut edges by doing 2-hop traversal along the cut edge. We have also proposed three strategies for graph partitioning that would help in indexing and querying.

The question that we try to address in this thesis is – "How can RIS handle very RDF graph?" Through our proposed design and strategies, we plan to partition YAGO2 [14] into small graphs, so that we can use RIS for indexing and querying.

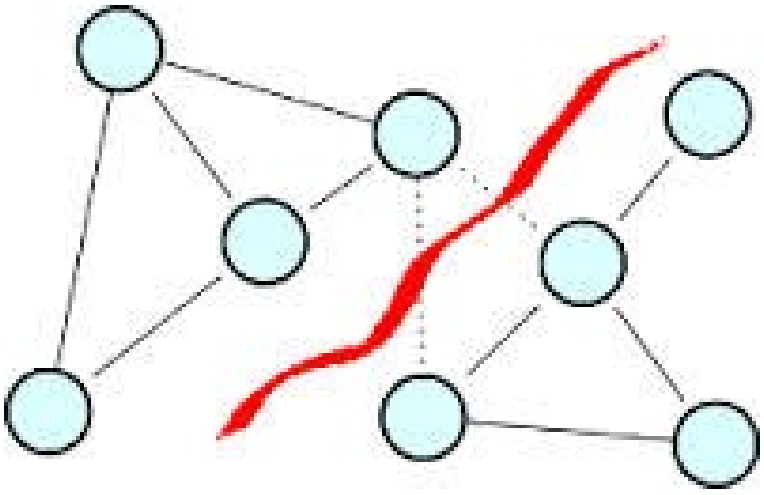


Figure 1. Example of cut-edge

1.2 Our Methodology

For all our experiments we have used YAGO2 [14] dataset, an extension of Yago knowledge base. It is very huge and highly-connected, which is a greater challenge for graph partitioning. We have tried many approaches such as finding out the cut edges and replicating them across the partitions they originate from, using different graph partitioning techniques to partition the graph, etc. This has led to an idea of having to reduce the edge-cuts, which means we will not lose many results while querying. So, our approach has many steps to handle the problem discussed in the Section 1.2, of which reducing the edge-cuts is the first. After partitioning the graph, we have a post processing step where we try to include the missing edges by 2-hop traversal followed by generation of connected components.

In order to avoid losing many results because of the graph partitioning, 2-hop traversal helps include the edges from the other partitions that can be reached by going up to 2-hops. This way we will not be losing results for all the queries that happen to span up to 2-hops across neighboring partitions. But it can be noted that due to 2-hop generation, we

obtain huge replication of the edges among many partitions leading to increase in the size of the actual partition because of the highly connectedness of the graphs that we are dealing with. We have plots in Chapter 5 that show the percentage increase in the triples in each of our strategies.

Due to the hugeness and highly–connectedness of YAGO2 dataset, some of the approaches crash because of the limitations of the memory, while for remaining approaches it would take months to finish the execution. We had to optimize our code several times, so that we would finish our execution in reasonable amount of time. We have plots in Chapter 5 that clearly state the time taken for graph partitioning for each strategy that we used.

CHAPTER 2

BACKGROUND

2.1 RDF

RDF (Resource Description Framework) [34] is a language used for representing information about Web resources in the World Wide Web. It is a data model proposed by the World Wide Web Consortium and has become standard for many popular datasets on the Internet. The resources represent not only web pages but also various things that are identified on the Web. It is easier to add arbitrary information about an entity or create link between two entities because of the schema-free nature of RDF.

RDF data is represented as a collection of <subject, predicate, object> triples. It also represents a graph data model where the adjacent nodes represent two things and the directed labeled edge denotes the relationship between those nodes. The subject in each triple is the entity from which the edge starts (source node), the object is the entity on the other side of the edge (sink node) and the predicate is the labeled edge. The subject and the predicate are always a URI whereas the object can be either a URI or a literal.

Any RDF data can be represented as a RDF graph and vice-versa. For example, Figure 2 is an example of a RDF graph. All the connected edges, which are the predicates, connect two nodes – the subject and the object. <*Mark, wasBornIn, Paris*> is one of the triples which can be represented as an edge named *wasBornIn* from the node *Mark* to the node *Paris*.

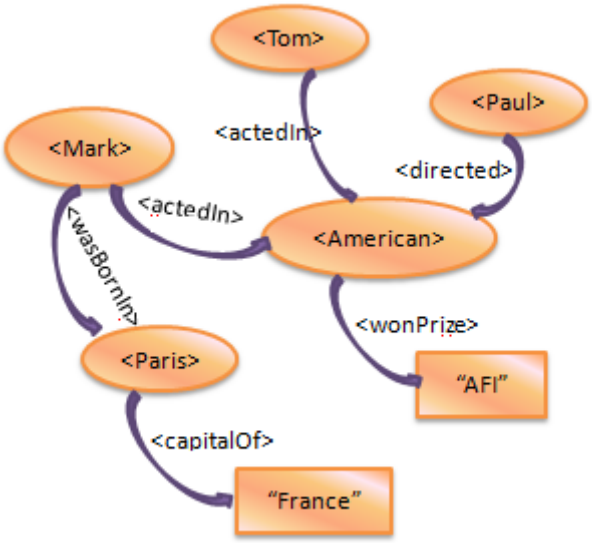


Figure 2. Example of RDF graph

2.2 SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) [39] is a popular query language that is used for querying RDF data. As stated in W3C Recommendation document – "Most forms of SPARQL queries contain a set of triple patterns called a basic graph pattern. Triple patterns are like RDF triples except that each of the subject, predicate and object can be a variable. A basic graph pattern matches a sub graph of the RDF data when RDF terms from that sub graph may be substituted for the variables."

For Example, the below query can be used to get the list of persons who acted in a movie that the won Prize *"AFI"*. When the below query is used to query the RDF data given in Figure 2, the result we get is <*Tom*> and <*Mark*> which is showed in Table 1. Figure 3 shows the graph representation of the SPARQL query.

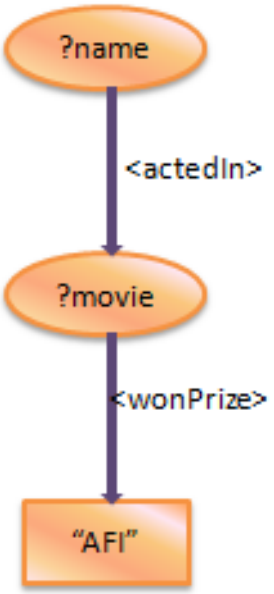


Figure 3. Example of SPARQL query

```
SELECT ?name
{
    ?name   <actedIn>    ?movie .
     ?movie  <wonPrize>   "AFI" .
}
```

Table 1. Result of RDF example

?name
<Tom>
<Mark>

2.3 Related Work

Today, there are a number of open-source and commercial tools for storing and querying large RDF graphs (e.g., 3Store [24], YARS2 [9], Jena [19], etc). These tools either store and process RDF in main memory or use RDBMS. The popular approach has been to use RDBMS by converting RDF triples into relational database systems, converting SPARQL queries to SQL queries and using these SQL queries on relational data. This was the approach followed by RDF stores like Vertical Partitioning [1], Hexastore [18], RDF-3X [38], BitMat [21], etc. RDF-3X and Hexastore have outperformed the vertical partitioning approach by storing the RDF data in a single triples table and building exhaustive indexes on the six permutations of (subject, predicate, object) triples.

CHAPTER 3

PROPOSED DESIGN

3.1 Overview

This chapter presents the design and implementation of the complete graph partitioning procedure including the different strategies used for partitioning. The graph partitioning procedure can be classified into three main steps – preprocessing step, partitioning using gpmetis [28] program and post processing step. Figure 9 shows the complete design of our proposed solution.

Preprocessing step consists of all the steps needed to generate the input accepted by gpmetis [28] program. The input for gpmetis [28] for a graph G = (V, E) with n vertices is a file with n+1 lines where first line has header information and remaining lines contain the information about each vertex of the graph. After we get the input file in the manner acceptable by gpmetis [28], we execute gpmetis [28] program. This gives us the partitioned graph – 'which vertex falls in which partition' information. Then, our post-processing step has many sub-steps depending on the strategy, which takes the output file generated by gpmetis [28], processes it to gets the actual n-tripled partitions. It also generated the list of edge-cuts. Here, we refer to edge-cut as the edge in the graph that is cut as a result of graph partitioning.

3.2 Considered Strategies

3.2.1 Strategy I

Here the main difference with the other strategies is that we use the complete dataset without any changes for our processing. We use the default options of gpmetis [28] i.e., no edge weights, no vertex weights.

3.2.2 Strategy II

In strategy II, we remove some type of edges from the original dataset resulting in the 'Reduced graph'. For YAGO2 dataset, the Reduced graph is obtained after removing triples that have rdf:type as an edge or literal as an object. Now, this reduced graph is used in our pre-processing step to generate the input file for gpmetis [28] as shown in Figure 4. Again, there are no edge weights or vertex weights in this strategy. We do keep track of the removed edges and add them back to the appropriate partitions in the post processing step.

3.2.3. Strategy III

This is done in the same manner as Strategy II, but while generating the input file for gpmetis [28] we also calculate vertex weights as shown in Figure 5. Thus, gpmetis [28] makes use of these vertex weights while generating the partitions. Vertex weights are calculated as the number of incoming and outgoing edges for that vertex. The weights are calculated on the original graph, which means along with the normal edges in reduced graph, it includes rdftype edges and edges connected to literals. We observed that the partitions obtained through this strategy are distributed uniformly when compared to the other strategies.

When we give weights to the vertices, we believe that we will get balanced partitions from gpmetis [28]. This is really important as we are already removing some of the triples before partitioning. When we add these extracted triples during post-processing step to these balanced partitions, we believe that the triples get uniformly distributed across all the partitions. Unbalanced partitions may cause I/O issues while indexing and querying as we may have very large partitions.

3.3. METIS

Metis is a software package for partitioning large irregular graphs, large meshes and computing fill-reducing orderings of sparse matrices. It was developed at the Department of Computer Science & Engineering at the University of Minnesota and is freely distributed. Metis provides numerous programs that can be used to partition graphs, partition meshes and also programs to convert meshes to graphs. It uses three phases for partitioning: graph coarsening, initial partitioning and uncoarsening. As our focus is on graph partitioning, we used a graph partitioning program called as gpmetis [28].

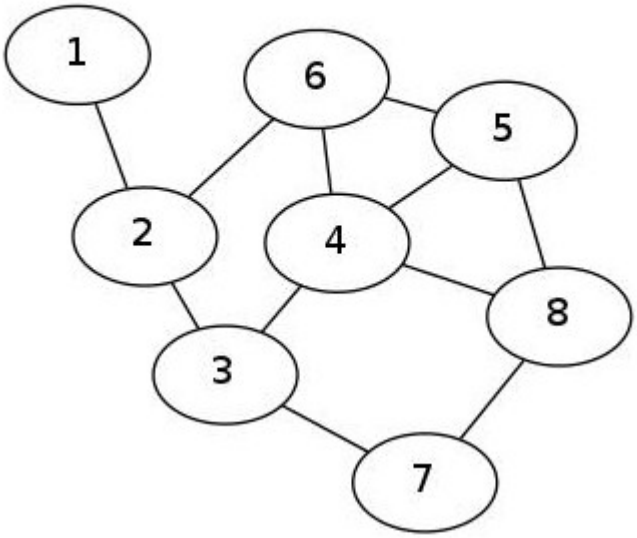
3.3.1 gpmetis

Gpmetis partitions a graph into specified number of parts. If the name of the input graph is metisfile then the output of gpmetis [28] is stored as metisfile.part.nparts where nparts is the number of parts the graph was partitioned into.

The input for this program is a undirected graph which is stored in a file as represented in Figure 5 through Figure 8. A graph G = (V,E) with n vertices and m edges is stored as a file containing n+1 lines where line k (k > 1) contains the information about the list of vertices, vertex k-1 is connected to. When k=1, which is the first line of the file, the line contains the header information about the file.

Figure 4 illustrates the input graph file format for gpmetis [28] with no vertex weights and edge weights. By default, the vertex weights and edge weights are equal to 1. The first line contains header information as n = 8, m = 11 and fmt is null. From line two are the connecting vertices listed in each line. Each line represents all the vertices connected to a particular vertex. For example, line two has all the vertices connected to vertex 1. If you look at the graph above, there's only one vertex connecting the vertex 1. So the second line of the

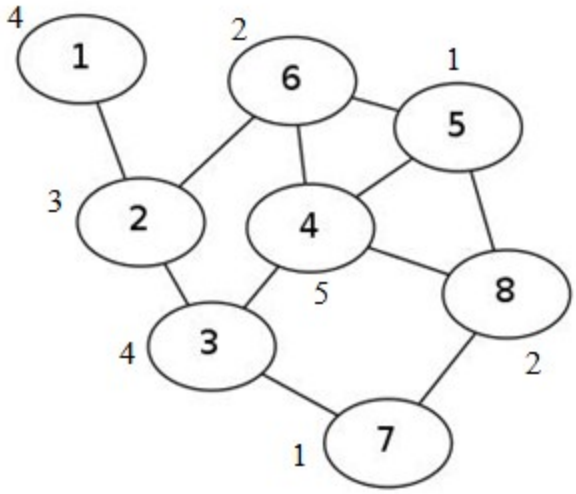
input is just the connecting vertex, i.e., 2. If you look at vertex 2, it has three connecting vertices which are 1, 3 & 6. Hence, the third line of the input contains those vertices which are connected to vertex 2. The connecting vertices can be written in any order.




```
8 11
2
1 6 3
2 4 7
6 5 8 3
6 4 8
2 4 5
3 8
4 7 5
```

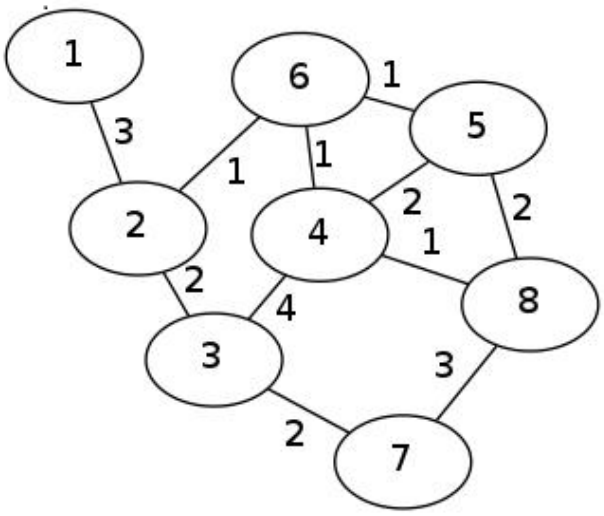
Figure 4. gpmetis input; $\Delta = 1$, $\Omega = 1$

Figure 5 illustrates the input graph file format for gpmetis [28] with different vertex weights and edge weights equal to 1. The first line as n = 8, m = 11 and fmt = 010, which means edge weight is 1 with variable vertex weight. fmt = 010 identifies that the graph has variable vertex weights. The line two through end represents the weight associated with corresponding vertex followed by the vertices connecting it. Every line starts with the weight




```
8 11 010
4 2
3 1 6 3
4 2 4 7
5 6 5 8 3
1 6 4 8
2 2 4 5
1 3 8
2 4 7 5
```

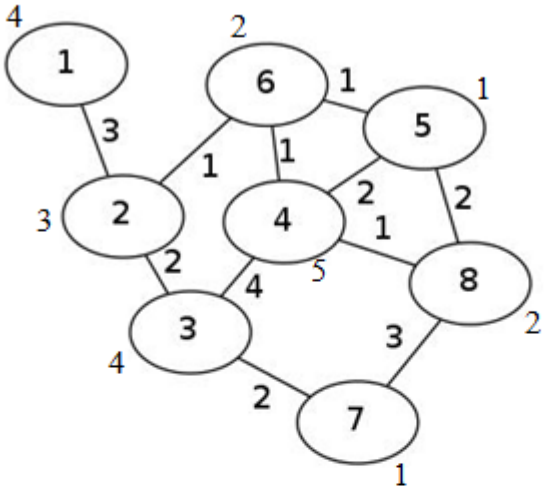
Figure 5. gpmetis input; $\Delta = \Theta$, $\Omega = 1$




```
8 11 001
2 3
1 3 6 1 3 2
2 2 4 4 7 2
6 1 5 2 8 1 3 4
6 1 4 2 8 2
2 1 4 1 5 1
3 2 8 3
4 1 7 3 5 2
```

Figure 6. gpmetis input; $\Delta = 1$, $\Omega = \Theta$

of that vertex. This is followed by the vertices connecting it. For example, consider the vertex 1. It has weight of 4 and is connected to vertex 2. So, the input line will be '4 2' which can be seen on the line two of the input above. If you look at vertex 2, it has weight 3 and there are three connecting vertices which are 1, 3 & 6. Hence, the third line of the input contains the weight '3' along with the three vertices which are connected to vertex 2. As the connecting vertices can be written in any order, the line three shows the input associated to vertex 2 as '3 1 6 3'.




```
8 11 011
4 2 3
3 1 3 6 1 3 2
4 2 2 4 4 7 2
5 6 1 5 2 8 1 3 4
1 6 1 4 2 8 2
2 2 1 4 1 5 1
1 3 2 8 3
2 4 1 7 3 5 2
```

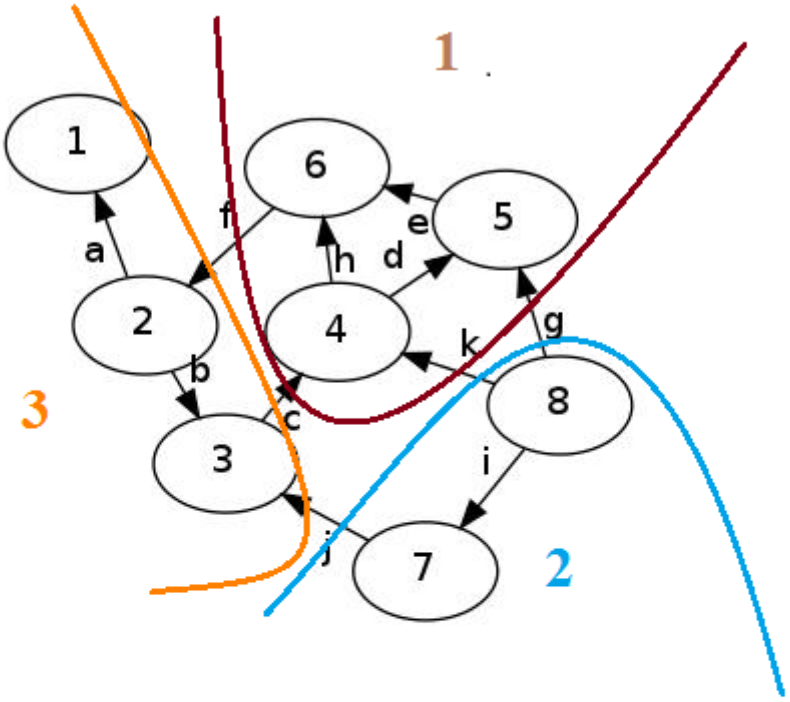
Figure 7. gpmetis input; $\Delta = \Theta$, $\Omega = \Theta$

Figure 6 illustrates the input graph file format for gpmetis [28] with different edge weights and vertex weights equal to 1. The first line as n = 8, m = 11 and fmt = 001, which means vertex weight is 1 with variable edge weight. From line two are the vertices which are connected to a vertex followed by each edge weight. The input line contains a connecting vertex first, which is then followed by its edge weight connecting the same vertex and so on. For example, consider vertex 1, which is connected only 1 vertex which is 2. The input in

this case will be 2(vertex) which is followed by 3(edge weight connecting 2). If you considervertex 2, which has multiple connecting vertices - 1, 3 & 6, there are 3 edge weights connecting those vertices. These weights are followed by each corresponding connecting vertex. The line three shows that input for vertex 2 which is 1 followed by its edge weight 3, 6 followed by its edge weight 1, 3 followed by its edge weight 2.

Figure 7 illustrates the input graph file format for gpmetis [28] with different vertex weights and edge weights. The first line as n = 8, m = 11 and fmt = 011, where 011 specifies that edge weight and vertex weight are both variable. From line two through end, both vertex and edge weights are used along with connecting vertices. The line starts with the vertex weight, followed by connecting vertex which is then followed by the edge weight connecting the same vertex. For example, consider the first vertex 1. It has vertex weight of 4 and is connected to vertex 2 through edge of weight 3. So the input will be 4(vertex weight) followed by 2(connecting edge) and then 3(edge weight). If there are multiple connecting vertices like in vertex 4, vertex weight will be followed by 1st connecting vertex and weight of edge connecting 1st vertex, which is then followed by 2nd connecting vertex and so on. Hence, the input will be '5 6 1 5 2 8 1 3 4'.

Figure 8 shows the example of output of gpmetis [28]. If gpmetis [28] partitions the graph as shown in the top part of the figure, then the output of gpmetis [28] that we get is shown in the bottom part of the figure. Each line represents the vertex of the graph and the number present at each line is the partition number where that vertex belongs to.




```
Line 1:    3
Line 2:    3
Line 3:    3
Line 4:    1
Line 5:    1
Line 6:    1
Line 7:    2
Line 8:    2
```

Figure 8. gpmetis output

3.4. Pre Processing Step

We observed that YAGO2 [14] dataset has few star-shaped graphs. Any graph whose maximum longest distance between any two nodes is not more than 2 can be called as a star-shaped graph. Initially, we have identified such star-shaped graphs and extracted them out of the dataset. We call this new dataset as ‘Reduced graph’. We treat each extracted star-shaped

graph as a partition and add them to our final list of partitions. Now, we use the reduced graph for all our processing going forward.

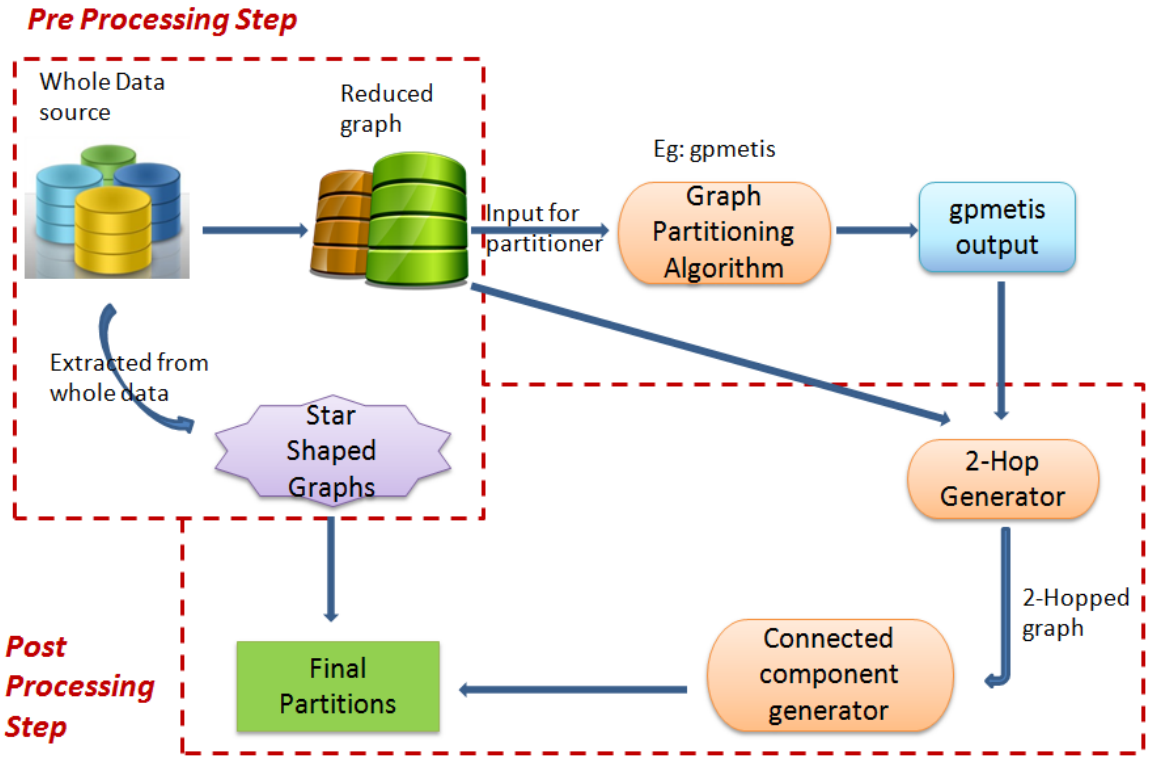


Figure 9. Proposed design

As it is difficult to work with ntriples when subjects, predicates and objects are in string format, we plan to convert them the dataset into ID-formatted dataset. So, during pre-processing step, the subjects, predicates and objects are extracted and a unique id is given to each of them. We use these unique IDs to replace the subjects, predicates and objects in the original graph. Now, it becomes easy to process ntriples with IDs as they consume less memory when stored in map, lookup takes less time, etc. we use IDs throughout the process of graph partitioning and finally after we get all the partitions, we replace these IDs with the actual subjects, predicates and objects. Once we get the files with ntriples with IDs, we start

generating the input for gpmetis [28]. Considering the huge size of the datasets, it becomes impossible to use maps that are placed in the main memory. Though it takes considerably long time accessing maps from the secondary memory, we need to place maps in secondary memory. This can be done using jdbm library. Figure 10 shows an example of an original graph which contains two star-shaped graphs and Figure 11 shows the reduced graph where those star-shaped graphs are removed.

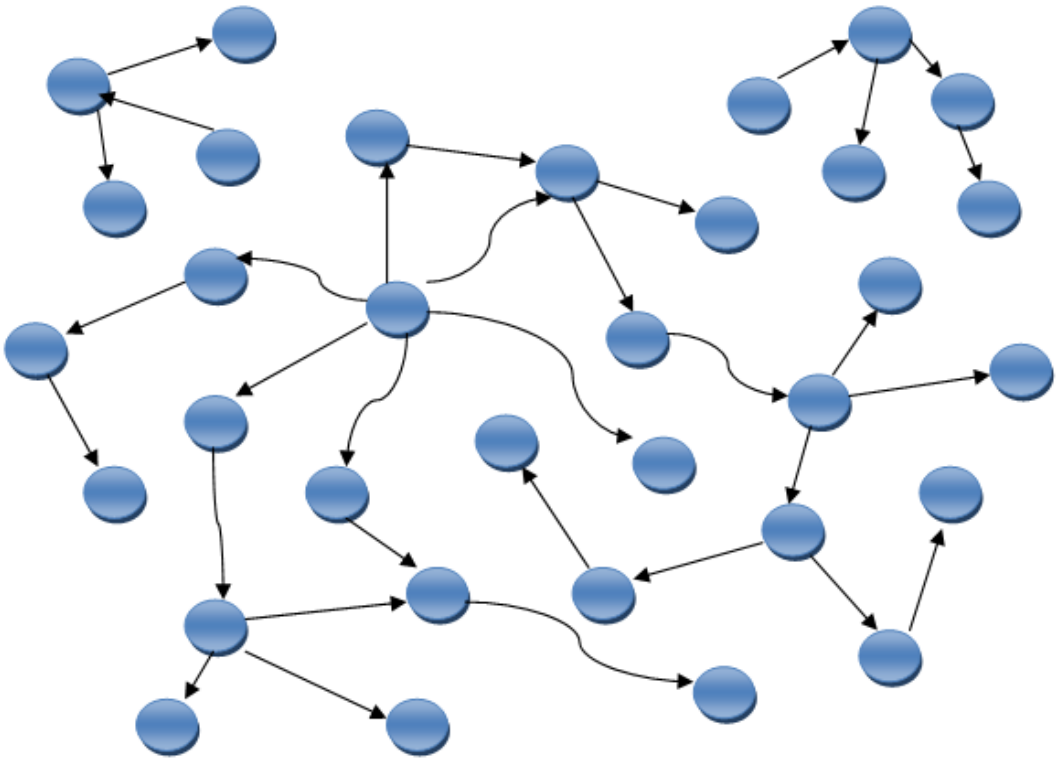


Figure 10. Original graph

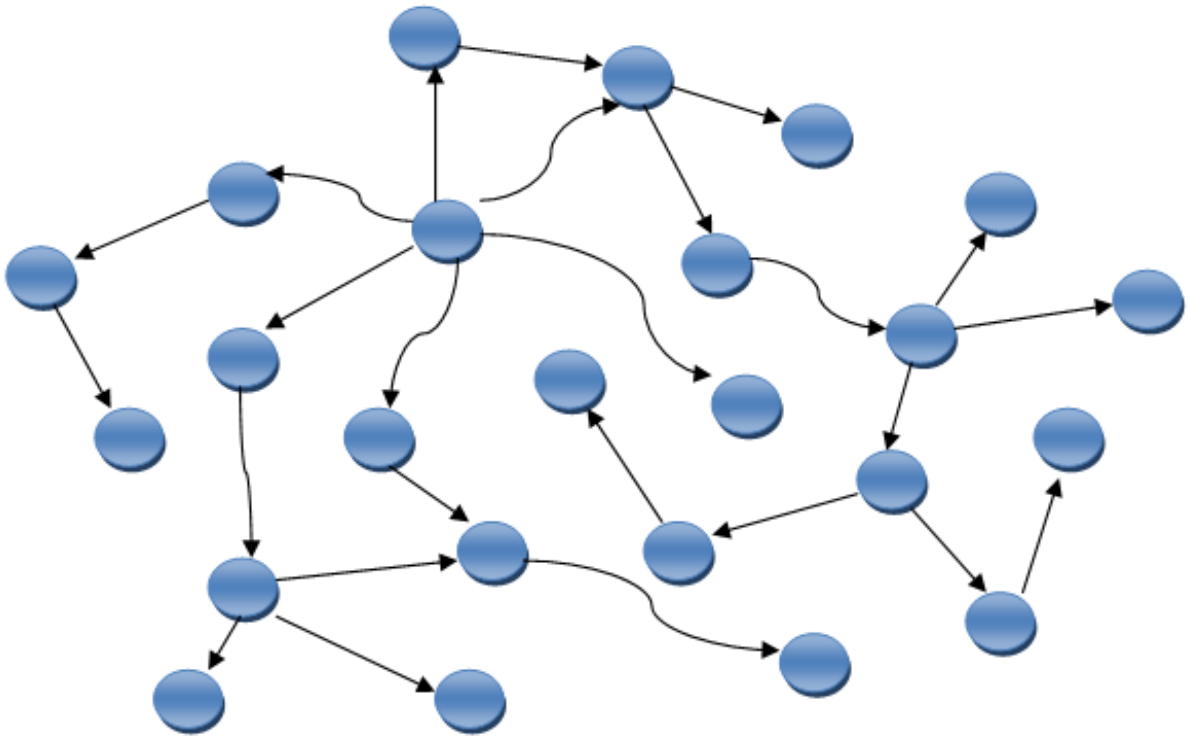


Figure 11. Reduced graph

3.5. Post Processing Step

The output of gpmetis [28] is processed to form the partitions. Figure 12 shows the output graph that we get when the reduced graph in Figure 11 is sent to gpmetis [28] for three partitions. This is done by looking at the list of vertices that fall into same partitions and check if there are any edges from a vertex in a partition to another vertex in the same partition. Here it can be noticed that the edges that got cut during graph partitioning are ignored.

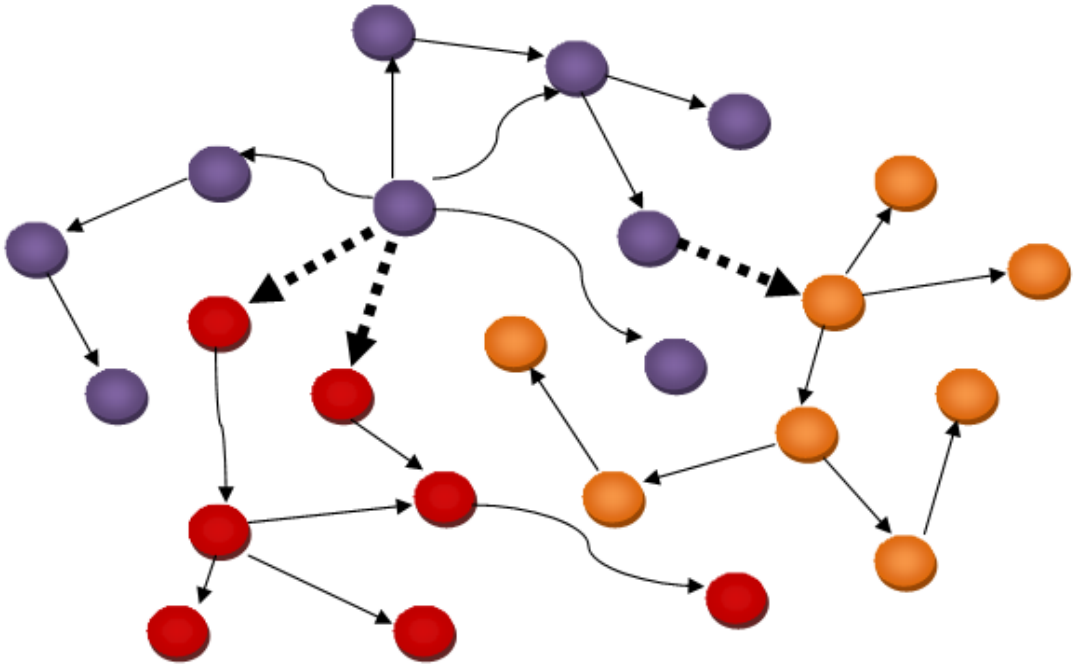


Figure 12. Partitioned graph

Next step in post-processing is 2-hop generation. Each partition is extended along the cut edges upto 2 hops. By this way, there is very possibility that each partition overlaps with its neighboring partitions along these edges that are included due to 2-hop generation, thus increasing the chances of query hit. But, it is possible that duplicate results are generated for specific queries as a result of this replication of data. This is a limitation of this approach.

Once 2-hop generation is completed, extraction of connected components among these partitions, if any is followed. If a partition has more than one connected component, then all those components are separated and treated as individual partitions henceforth. If the partition is completely connected and thus has only one component, it is treated as one partition like before. Figure 13 shows the example of 2-hop traversal along on the cut-edges of one of the partitions showed in Figure 12.

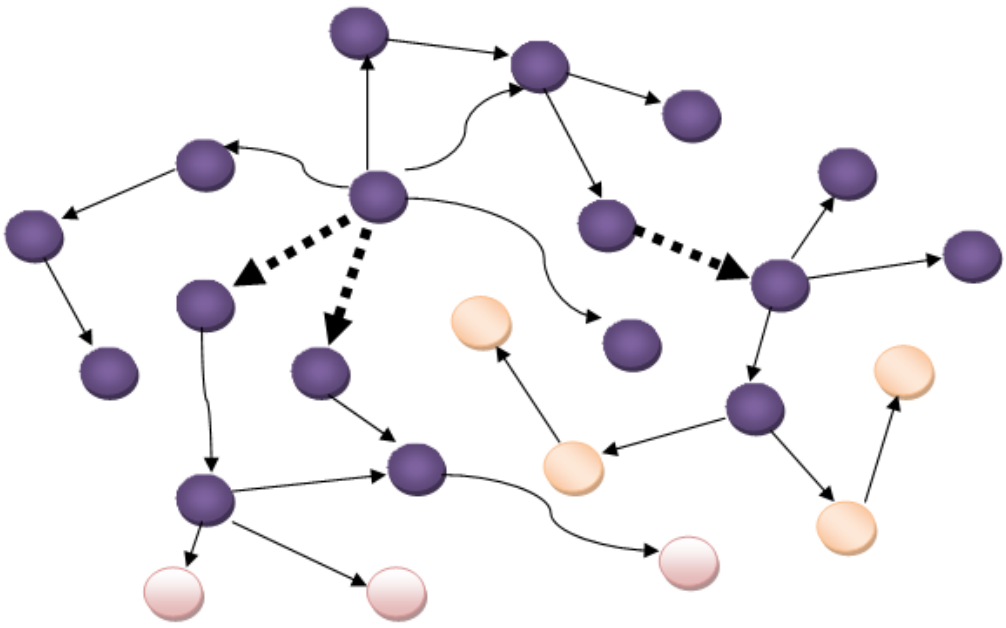


Figure 13. 2-hop traversal

Now, all triples with rdf:type as predicate and all triples with a literal as an object are added back to these individual partitions. Each vertex in every partition is analyzed to see if it had any edge with rdf:type predicate or object as literal that was removed previously. Thus, all these removed edges are added back to the vertex they are part-of. This way, no edge is removed permanently from the graph even after the partitioning process. Thus, the removed edges would not have actively participated in the graph partitioning process but be partitioned among the resulting partitions.

Final step of post processing is conversion of IDs to the actual strings. The map used to convert string to IDs is used again to write the IDs back to the strings. At the end of this step, the final partitions of the huge dataset are produced after going through the long graph-partitioning process.

CHAPTER 4

IMPLEMENTATION

4.1 Data Structure

Throughout our process, there are numerous times when we need data structures for processing. Mainly we need them to store and retrieve the data temporarily. We could very well use traditional maps for this. But, the data we are dealing with is very huge and thus it cannot scale more than the main memory because maps can be stored only in main memory. Hence there is need for a better data structure which is more persistent and where the data can be stored on secondary memory as the data we are dealing with is too big for the main memory.

Jdbm2 package [42] is a solution for our problem. It is a java package that contains public API. It has many interfaces, classes and methods that allow us to store data persistently on secondary memory.

RecordManager is an interface to manage records, which are objects that are serialized to byte[] in the background. It is responsible for handling transactions. It is also a factory for all the primary maps. RecordManagerFactory is a factory class that can be used for instantiating RecordManager instances. It has createRecordManager method that is used to create RecordManager objects. It takes some location on the hard disk as the parameter.

PrimaryTreeMap, PrimaryTreeMap and PrimaryStoreMap implements java.util.map interface that has all traditional maps. These data structures use page store for persistence. So, Hash maps and Tree maps can be created with billions of items but more emphasis is on regular commits.

```
import jdbm.PrimaryHashMap;
import jdbm.RecordManager;
import jdbm.RecordManagerFactory;

public class class1 {
  public void method1(){
    try {
      // Get Manager singleton
      RecordManager recman =
RecordManagerFactory.createRecordManager("/home/db985/Database
/testdata");

      //create persistant hashmap
      PrimaryHashMap tMap=
recman.hashMap("/home/db985/Database/testmap");

      //Store data into the map
      while(...) {
        tMap.put();
        //Commit after regularly intervals
        recman.commit();
      }
      //Commit after the last insert
      recman.commit();

      //Use tMap
      while(...) {
        tMap.get();
      }

      recman.close();
    }
    catch (Exception e){
    }
  }
}
```

Figure 14. Sample code snippet for using PrimaryHashMap()

When to make commits can be quite challenging in case of persistent storage. Making a call to Commit() after every put() command can be an overhead in terms of performance. Similarly, it cannot be called only once after storing all data (assuming we are dealing with very huge data) because it then runs out of memory like java collection maps. So, it is better to make a call to commit() at regular intervals.

CHAPTER 5

PERFORMANCE EVALUATION

We conducted a comprehensive study of all the strategies of graph partitioning described above and the results are reported in this section. We ran the experiments on a single quad core machine with 16 GB RAM. We used C++ and java languages for coding.

5.1 Evaluation Metrics

We used YAGO2 [14] dataset for our evaluations. In order to compare the three strategies mentioned in the paper, we measured (a) time taken for each strategy to partition the dataset into 500, 1000 and 2000 partitions, (b) number of edge-cuts (c) relative growth of dataset for each strategy. Each of these experiments is done for different values of 'vertex degree'.

YAGO2 [14] dataset is an extension of YAGO knowledge base that focuses on spatial and temporal knowledge. It represents general world knowledge built from Wikipedia, WordNet and GeoNames, and contains nearly 10 million entities and events.

Table 2. Full YAGO2 dataset

Total # of subjects	**Total # of predicates**	**Total # of objects**
32,012,507	94	28,984,072

Table 3. # of triples and vertices in YAGO2 dataset

Total # of triples (aka. Edges)	**Total # of vertices**
195,048,654	29,484,037

Table 4. Total time taken for complete graph partitioning

	#Partitions = 500 (time in hours)		#Partitions = 1000 (time in hours)	
VertexDegree	**T2**	**T3**	**T2**	**T3**
120000	214.8	117.1	240.8	151.76
100000	110.37	105.4	121.21	120.37

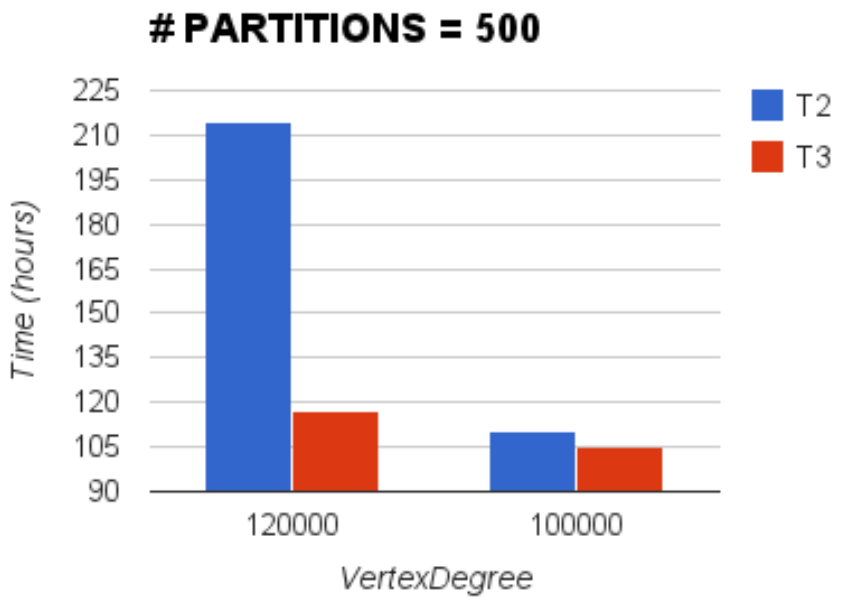


Figure 15. Total time taken to partition YAGO2 into 500 partitions

Figure 15 and Figure 16 show the time taken to partition YAGO2 into 500 and 1000 partitions respectively. Clearly it can be seen that the time taken for T3 is less than that of T2. The reason for this behavior is because of the balanced partitions in T3 than in T2. 2-hop traversal time increases as the size of the partition increases. Table 4 gives the actual timings for the graph partitioning.

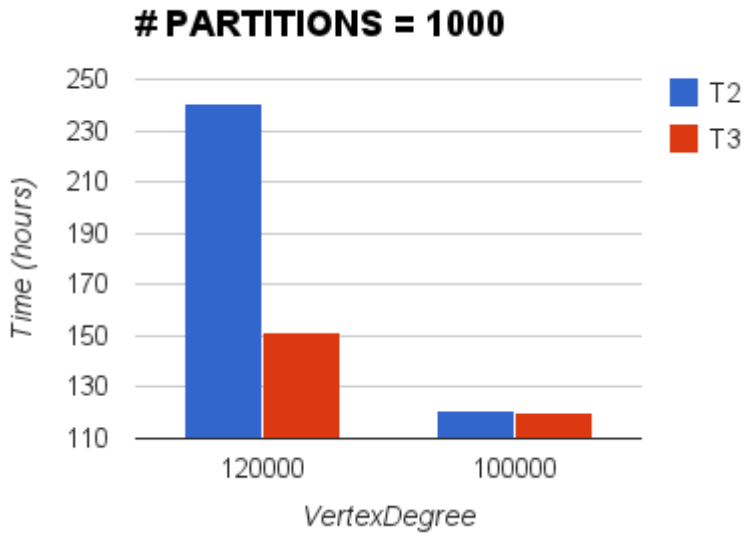


Figure 16. Total time taken to partition YAGO2 into 1000 partitions

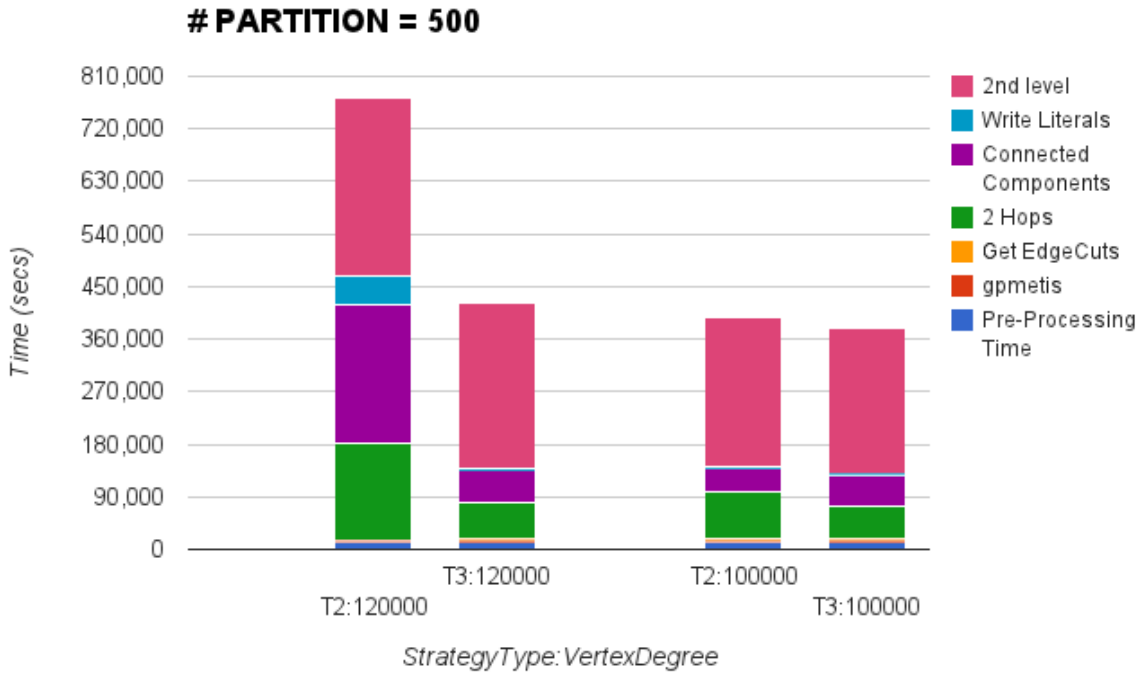


Figure 17. Time taken for each step when # Partitions = 500

Table 5. Time taken for each step during graph partitioning for #Partitions = 500

Strategy Type :Vertex Degree	**Pre-processing (secs)**	**Partitioning Time (secs)**	**Post Processing Time (secs)**				**2nd Level (secs)**
			Get Edge cuts	*2-Hop Generation*	*Connected Components*	*Converting id to string*	
T2 : 120000	12,637	12,730	260	166,695	235,748	49,387	305,822
T3 : 120000	11,833	3,981	2,371	61,465	54,679	3,660	283,570
T2 : 100000	12,637	2,849	2,416	79,248	40,549	2,957	256,677
T3 : 100000	11,833	3,961	2,232	54,517	54,475	3,788	248,645

Table 6. Time taken for each step during graph partitioning for #Partitions = 1000

Strategy Type :Vertex Degree	**Pre-processing (secs)**	**Partitioning Time (secs)**	**Post Processing Time (secs)**				**2nd Level (secs)**
			Get EdgeCuts	*2-Hop Generation*	*Connected Components*	*Converting id to string*	
T2 : 120000	12,637	3,046	260	166,695	257,774	50,442	376032
T3 : 120000	11,833	3,851	2,430	46,032	115,590	7,432	359174
T2 : 100000	12,637	3,046	2,190	68,329	50,591	3,397	296154
T3 : 100000	11,833	3,766	3,079	41,814	60,225	4,087	308542

Figure 17 shows the stacked graph for #Partitions = 500 with StrategyType:VertexType on x-axis and time taken by various steps in the process on y-axis. The graph clearly shows that the 2-Hop generation and generation of connected components

takes more time when compared to other steps in the process. Table 5 has the split of the actual timings corresponding to the Figure 17.

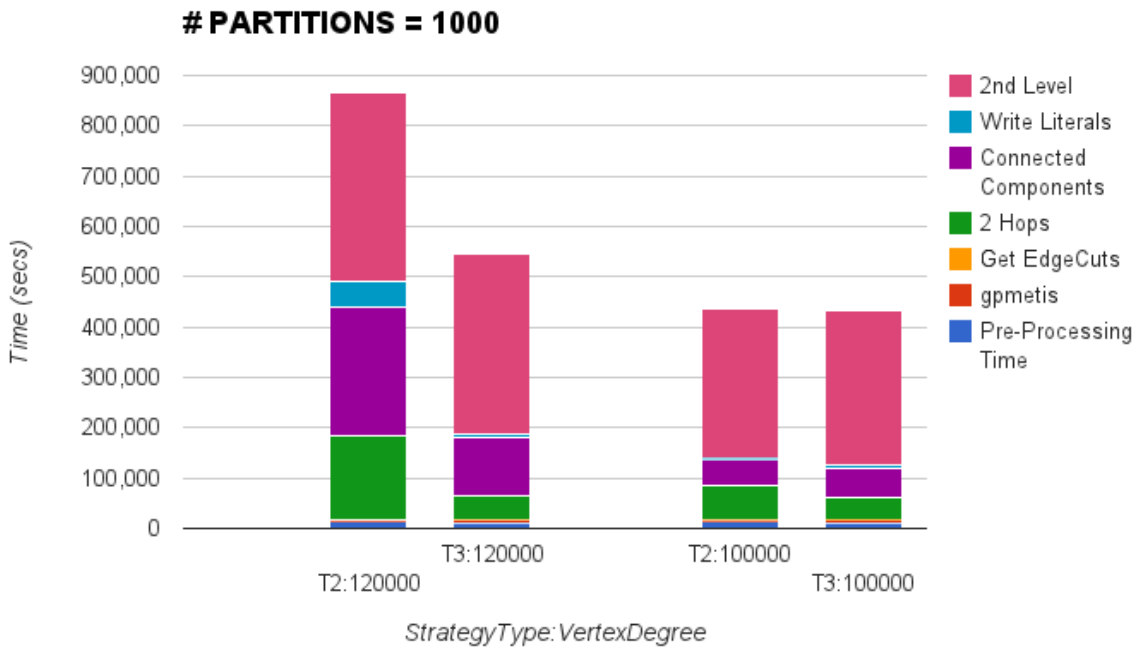


Figure 18. Time taken for each step when # Partitions = 1000

Figure 18 shows the stacked graph for #Partitions = 1000 with StrategyType:VertexType on x-axis and time taken by various steps in the process on y-axis. As shown in the graph, the 2-Hop generation and generation of connected components takes more time when compared to other steps in the process. Table 6 has the split of the actual numbers correspoding to the figure 18.

Table 7. Total # of edge cuts

Strategy Type	#Partitions = 500	#Partitions = 1000
T2	17,403,516	19,684,314
T3	20,999,420	23,442,696

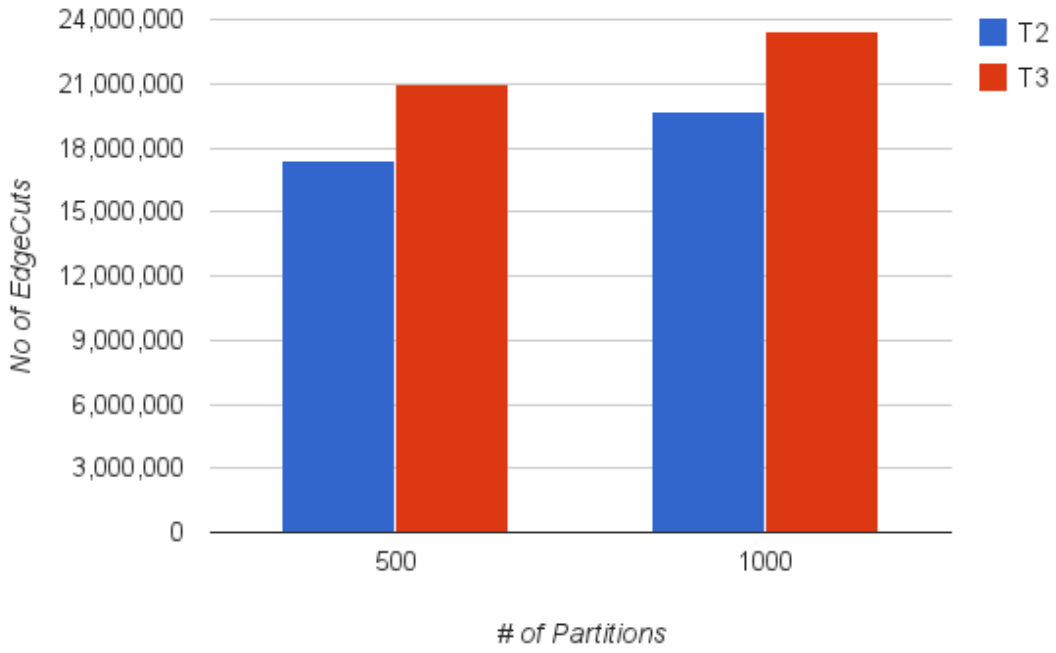


Figure 19. Number of edgecuts for each strategy for both 500 and 1000 partitions

Figure 19 represents the # of edgecuts for each strategy for 500 partitions and 1000 partitions. From the graph, it can be clearly understood that as the number of partitions increase, we tend to cut more number of edges. When we compare strategy T2 with T3, T3 has more number of edge cuts than T2 because in strategy T3 the number of triples across all

the partitions tends are uniformly distributed. In order to achieve this, gpmetis [28] cuts more edges than in T2.

Table 8. Percentage increase in # of triples

	#Partitions = 500		#Partitions = 1000	
VertexDegree	**T2**	**T3**	**T2**	**T3**
100000	297.1	342.47	345.15	417.51
120000	301.62	344.65	383.54	548.47

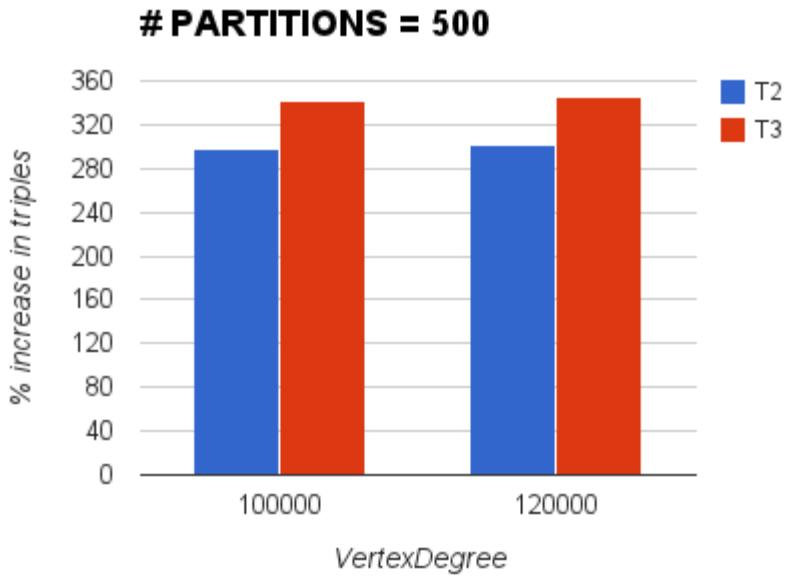


Figure 20. Percentage increase in # of triples for #Partitions = 500

Figure 20 and Figure 21 represents the percentage increase in number of triples for #Partitions = 500 and #Partitions = 1000 respectively. Across both the graphs it can be seen that the increase is more in strategy T3, as there are more number of edgecuts. During 2-hop traversal, we traverse along each cut-edge and include all the triples that can be reached from the other partitions.

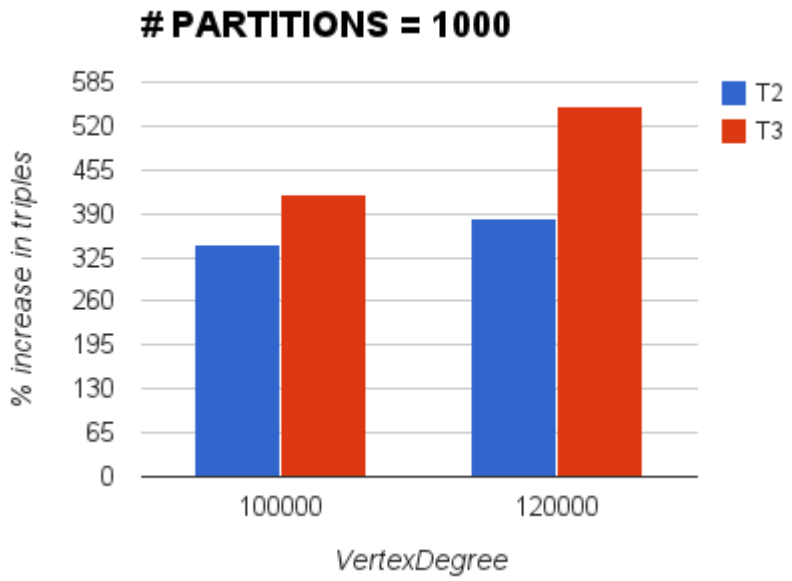


Figure 21. Percentage increase in # triples for #Partitions = 1000

Figure 22 and figure 23 show the time taken to index 500 and 1000 partitions across both the strategies T2 and T3 for vertexdegrees 100000 and 120000, respectively. Here it should be noted that y-axis is on logarithm scale. We used RIS indexing mechanism for indexing our partitioned graph and we also perform a comparison of our mechanism with RDF-3X. The time taken to index the paritions obtained for strategy T2 is less than that of strategy T3 for each vertexdegree. Indexing time of RDF-3X performs extremely well when compared to RIS indexing.

Table 9. Total time taken for indexing

	#Partitions = 500 (time in secs)		**#Partitions = 1000 (time in secs)**	
VertexDegree	**T2**	**T3**	**T2**	**T3**
100000	7068.31	74742.9	25426.31	222351
120000	12562.24	85261.1	30753.84	159882.1

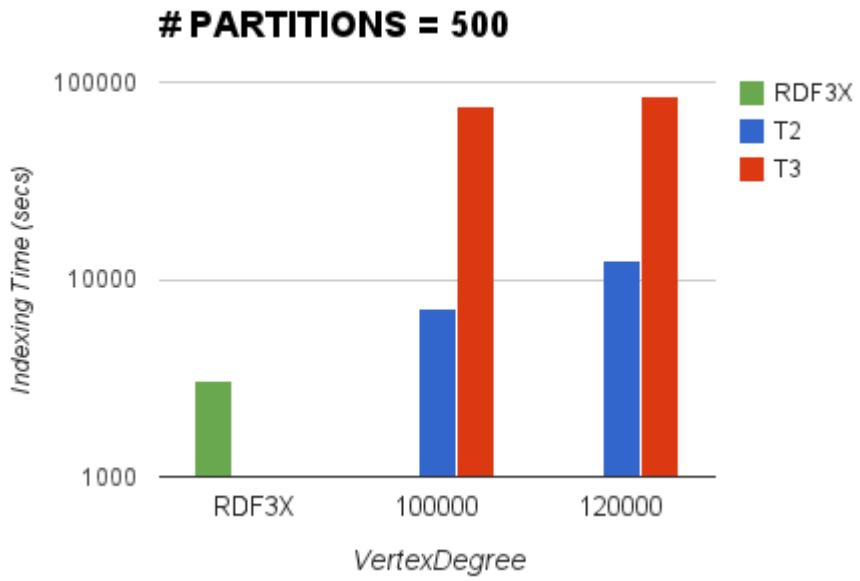


Figure 22. Time taken to index 500 partitions compared to RDF-3X

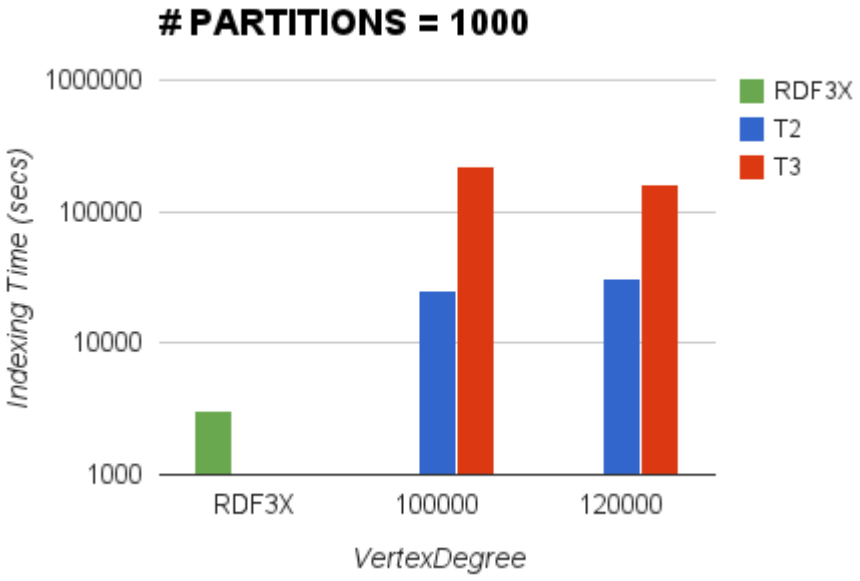


Figure 23. Time taken to index 1000 partitions compared to RDF-3X

For querying our partitions, we designed three queries keeping in mind to have more number of joins. RIS indexed graphs are more beneficial for very large queries with more number of joins. So, all our queries have more than 17 joins each. We have included all the queries that we used, in the section 4.2. We also have compared the query time of both the strategies with RDF-3X, which can be seen from figure 23 through figure 28.

Table 10. Querying time for Q1

	#Partitions = 500 (time in secs)		**#Partitions = 1000 (time in secs)**	
VertexDegree	**T2**	**T3**	**T2**	**T3**
100000	6.786	27.699	12.656	64.554
120000	8.395	31.288	17.246	99.143

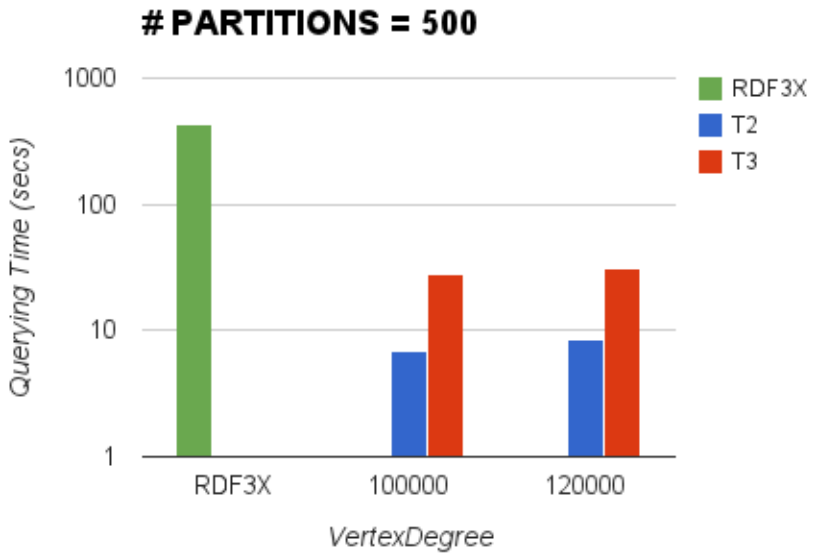


Figure 24. Time taken to query Q1 when #Partitions = 500 compared to RDF-3X

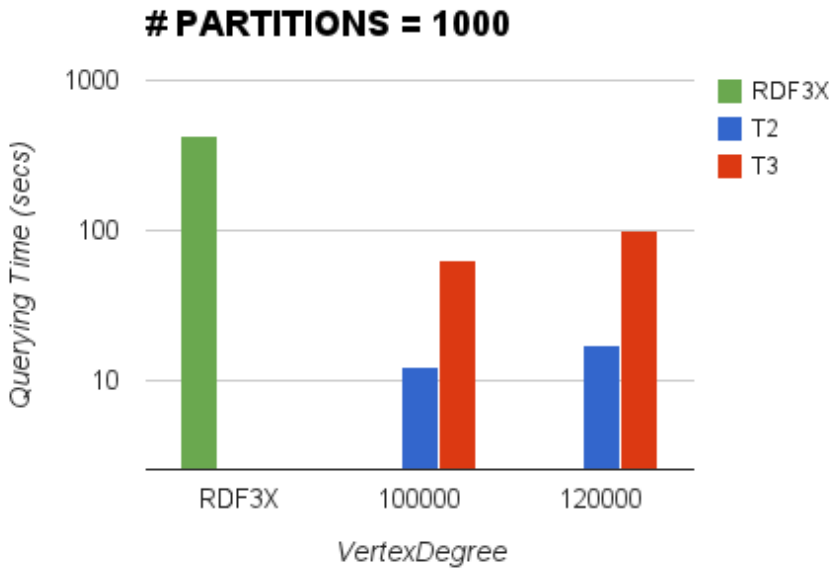


Figure 25. Time taken to query Q1 when #Partitions = 1000 compared to RDF-3X

Table 11. Querying time for Q2

	#Partitions = 500 (time in secs)		#Partitions = 1000 (time in secs)	
VertexDegree	**T2**	**T3**	**T2**	**T3**
100000	4.856	11.066	4.343	9.808
120000	4.921	11.831	5.021	14.593

Table 12. Querying time for Q3

	#Partitions = 500 (time in secs)		#Partitions = 1000 (time in secs)	
VertexDegree	**T2**	**T3**	**T2**	**T3**
100000	4.006	8.606	8.743	16.393
120000	5.734	7.387	9.382	16.987

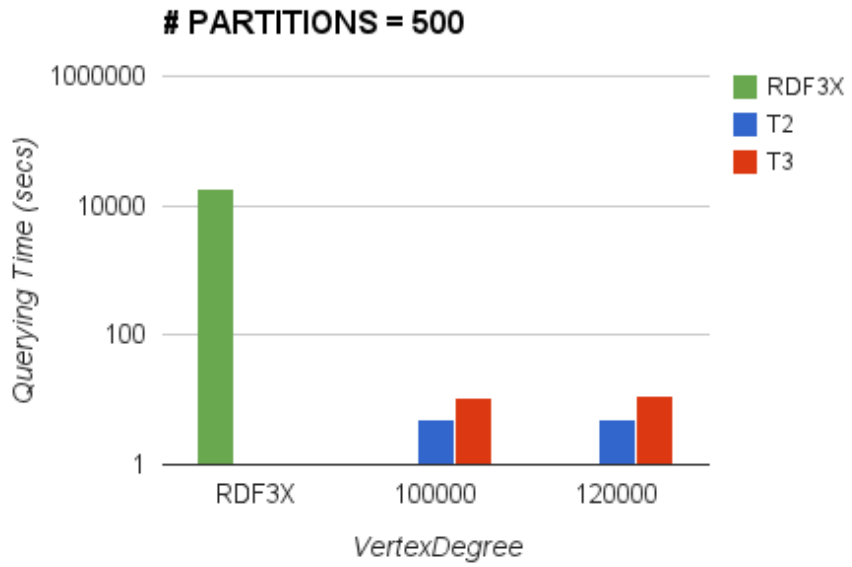


Figure 26. Time taken to query Q2 when #Partitions = 500 compared to RDF-3X

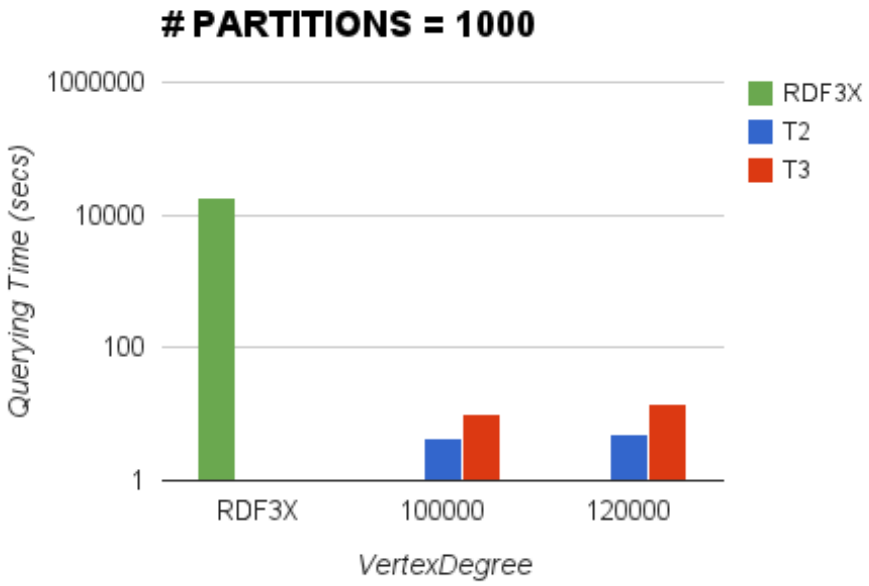


Figure 27. Time taken to query Q2 when #Partitions = 1000 compared to RDF-3X

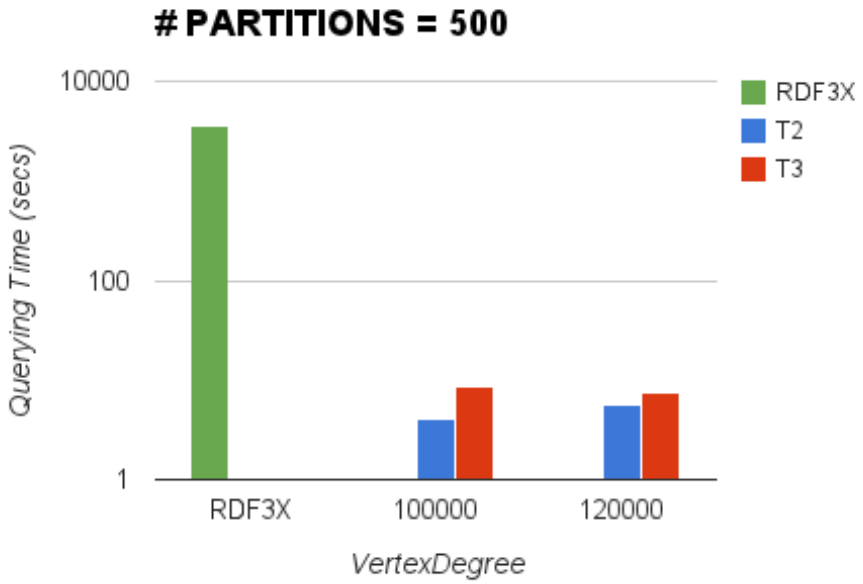


Figure 28. Time taken to query Q3 when #Partitions = 500 compared to RDF-3X

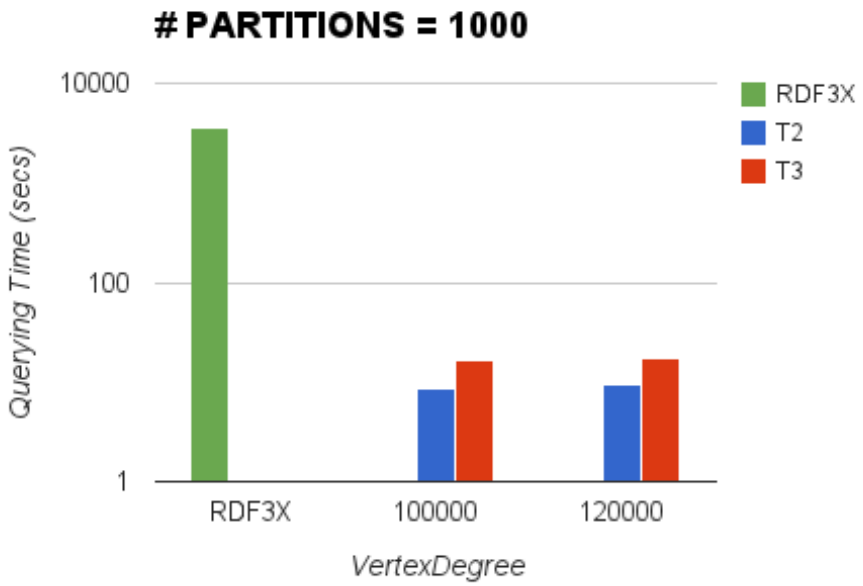


Figure 29. Time taken to query Q3 when #Partitions = 1000 compared to RDF-3X

Figure 24 and figure 25 illustrates the time taken to query Q1 for partitions 500 and 1000 respectively. In both the plots, it can be seen that T2 perfoms better than T3 but it should be noted that the there are more number of results that we miss in strategy T2 than strategy T3 when compared to RDF-3X. So, we still need to do further investigation on how to handle the cut edges.

Similarly, Figure 26 and figure 27 show the time taken to query Q2 for partitions 500 and 1000 respectively. And figure 28 and figure 29 represent the time taken to query Q3 for paritions 500 and 1000 respectively. In all the graphs, though T2 performs better than T3, the number of results we get in T3 is more than T2. Table 5 shows the number of joins each of the queries Q1, Q2 and Q3 have.

Table 13 shows the number of joins that were present in each of the queries that we used. Section 5.2 has the list of SPARQL queries. Figures 30 through Figure 32 show the graphical representation of the queries for better understanding.

Table 13. # of joins in each query

Query	# of joins
Q1	23
Q2	18
Q3	19

5.2 Queries Used

Query 1:

PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX y:<http://www.mpii.de/yago/resource/>
PREFIX x:<http://www.w3.org/2001/XMLSchema#>
SELECT ?Zita ?city ?link1
WHERE
{
    ?Zita y:diedIn ?city .
    ?P1 y:wasBornIn ?city .
    ?P2 y:wasBornIn ?city .
    ?city y:hasGeonamesId <http://www.mpii.de/yago/resource/3174530> .
    ?city y:participatedIn ?event1 .
    ?city y:participatedIn ?event2 .
    ?state1 y:hasCapital ?city .
    ?state2 y:hasCapital ?city .
    ?Zita y:hasInternalWikipediaLinkTo ?city .
    ?Zita y:hasInternalWikipediaLinkTo ?link1 .
    ?city y:hasInternalWikipediaLinkTo ?Zita .
    ?link2 y:hasInternalWikipediaLinkTo ?Zita .
    ?Zita y:hasInternalWikipediaLinkTo ?link2 .
    ?link2 y:hasExternalWikipediaLinkTo ?ext1 .
    ?x y:isCitizenOf <http://www.mpii.de/yago/resource/Italy> .
    ?event1 y:happenedIn ?link1 .
    ?x y:livesIn ?link1 .
```

```
    ?z y:isKnownFor ?link1 .
    ?x y:diedIn ?city2 .
    ?event1 y:happenedIn ?city3 .
}
```

Query 2:

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX y:<http://www.mpii.de/yago/resource/>
PREFIX x:<http://www.w3.org/2001/XMLSchema#>
SELECT ?p1 ?place
WHERE
{
    ?p1 y:hasWonPrize <http://www.mpii.de/yago/resource/Prix_Femina> .
    ?PRIZE y:hasInternalWikipediaLinkTo ?p1 .
    ?p1 y:hasInternalWikipediaLinkTo ?p5 .
    ?p5 y:hasInternalWikipediaLinkTo ?Univ .
    ?p1 y:hasInternalWikipediaLinkTo ?Univ .
    <http://www.mpii.de/yago/resource/Claude_Bernard> y:influences ?p6 .
    ?p6 y:worksAt ?Univ .
    ?Univ y:hasInternalWikipediaLinkTo ?place .
    ?p1 y:hasInternalWikipediaLinkTo ?place .
    ?p2 y:hasChild
    <http://www.mpii.de/yago/resource/Edmund_of_Woodstock,_1st_Earl_of_Kent> .
```

?p2 y:isMarriedTo ?p4 .

?p2 y:wasBornIn ?place .

?place y:hasInternalWikipediaLinkTo ?p3 .

?p3 y:influences <http://www.mpii.de/yago/resource/Louis-Ferdinand_C%C3%A9line>

.

?event1 y:happenedIn ?place .

?event1 y:hasInternalWikipediaLinkTo <http://www.mpii.de/yago/resource/England> .

<http://www.mpii.de/yago/resource/Jacquerie> y:hasInternalWikipediaLinkTo ?event1 .

?place y:hasGeonamesId <http://www.mpii.de/yago/resource/2968815> .

}

Query 3:

PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>

PREFIX y:<http://www.mpii.de/yago/resource/>

PREFIX x:<http://www.w3.org/2001/XMLSchema#>

SELECT ?p1 ?name2

WHERE

{

?p1 y:wasBornIn ?city .

?event1 y:happenedIn ?city .

?p2 y:diedIn ?city .

?p2 y:wasBornIn ?city .

?stud1 y:hasAcademicAdvisor ?p1 .

?p1 y:hasAcademicAdvisor ?p5 .

?p1 y:isKnownFor ?name .

?city4 y:hasInternalWikipediaLinkTo ?city .

?Jose y:hasInternalWikipediaLinkTo ?city .

?Jose y:wasBornIn ?city4 .

?Jose y:hasWonPrize <http://www.mpii.de/yago/resource/Prix_de_Rome> .

?p4 y:hasInternalWikipediaLinkTo ?event1 .

?name2 y:hasInternalWikipediaLinkTo ?city .

?name2 y:hasInternalWikipediaLinkTo ?Fran .

?Verm y:hasInternalWikipediaLinkTo ?Fran .

?Verm y:hasGender <http://www.mpii.de/yago/resource/male> .

?Verm y:participatedIn <http://www.mpii.de/yago/resource/First_Crusade> .

<http://www.mpii.de/yago/resource/Battle_of_Hallue> y:happenedIn ?Fran .

}

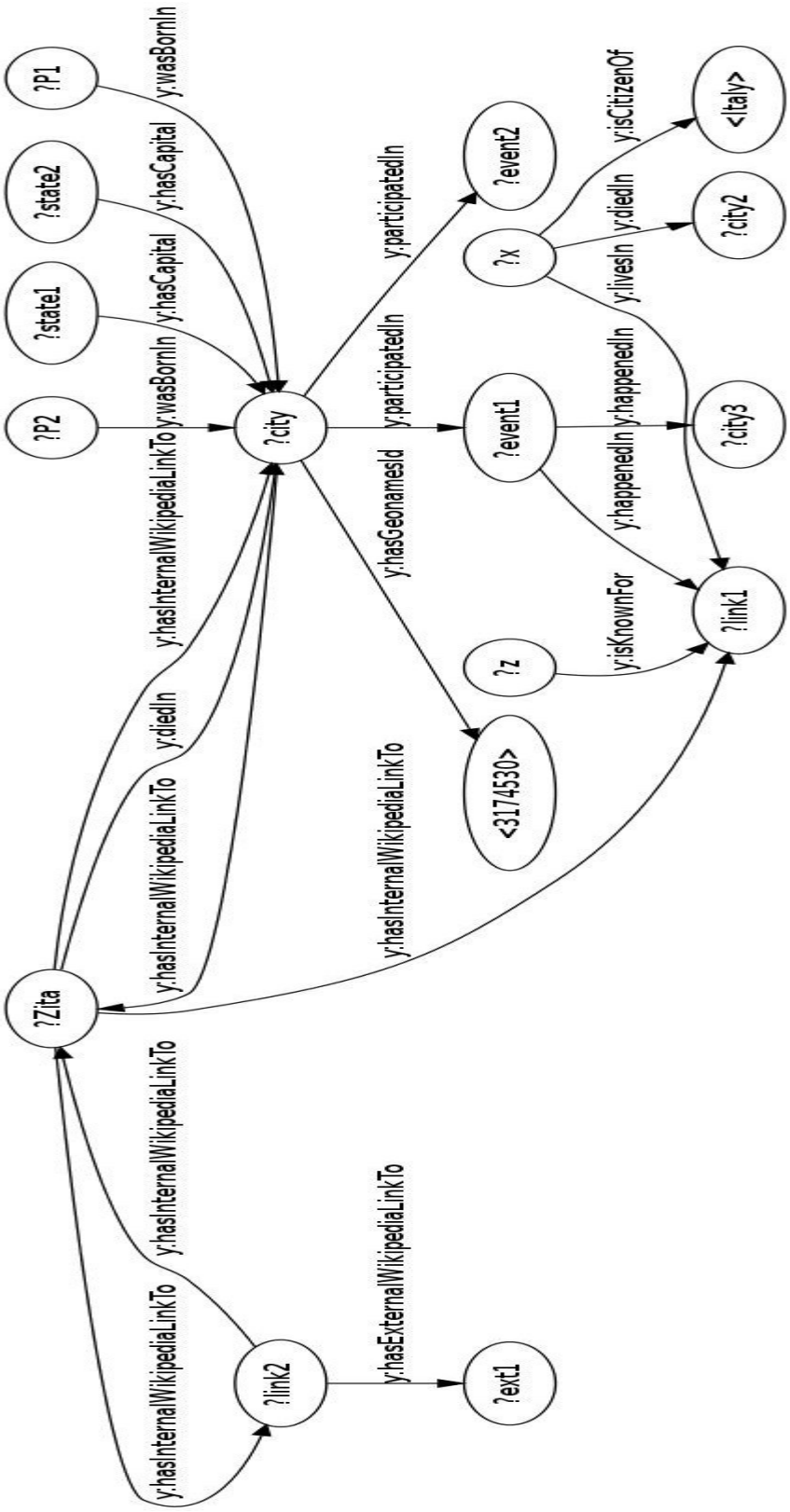


Figure 30. Graphical representation of query 1

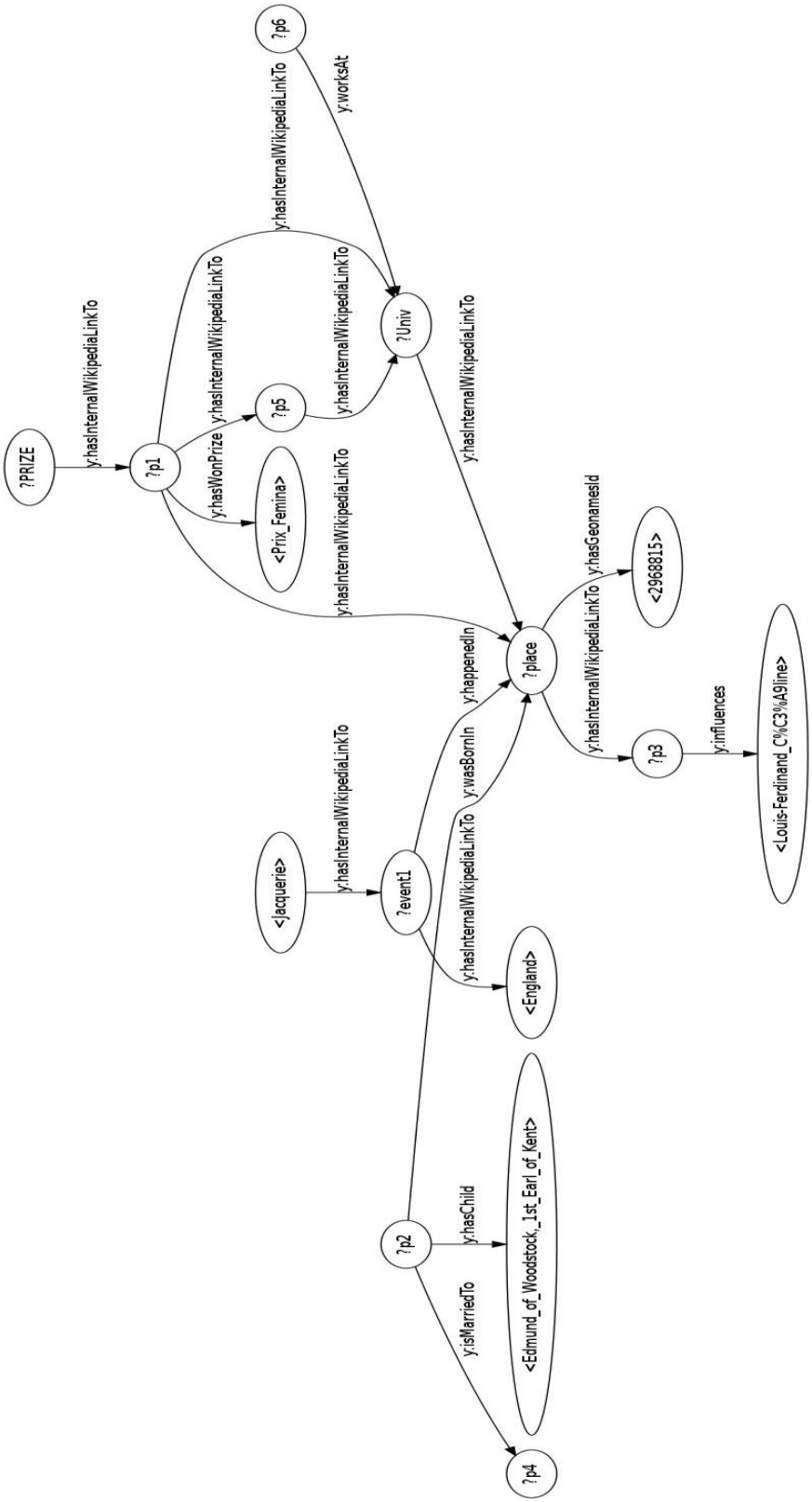


Figure 31. Graphical Representation of query 2

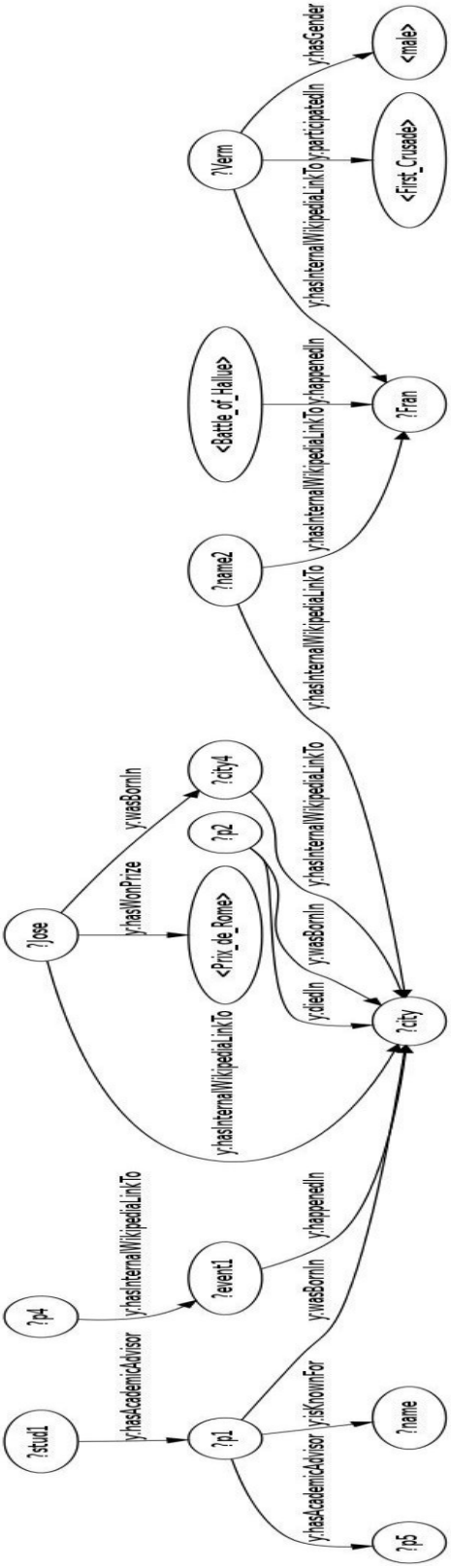


Figure 32. Graphical representation of query 3

5.3 Discussion

From our performance evaluations, though we observe that the time taken for all the queries (Q1, Q2 and Q3) for RIS is less than that of RDF-3X, we really do not feel we outperformed them because the number of results that we got for RIS is not equal to that of RDF-3X. We observed that we missed some results for both strategies T2 and T3. But, number of results in T3 are better than T2. So, the reasons why we have missed the results are:

1. Threshold for high degree vertex during 2-hop traversal being 100,000 and 120,000. So, we have stopped traversing the path whenever we encounter a vertex with degree more than the threshold.
2. The query diameter for all our queries is greater than 2. So, we might need to do more than 2-hop traversal to get all the results.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

We presented a study of two graph partitioning techniques for efficiently querying the huge RDF graph (i.e., YAGO2 [14]). We have demonstrated our design that tries to minimize the number of missing results due to edge-cuts, by 2-hop traversal along the cut edges. We also explained how we have reduced the number of edge-cuts by separating the edges with predicate as rdf:type and triples with literals as object. We have compared the two strategies T2 and T3 by indexing and querying the partitioned YAGO2 dataset using RIS and showed that partitioning the weighted graph is better than the un-weighted graph. We have also done hierarchical partitioning so that the partitions are suitable for RIS to run.

6.2 Future Work

For future work, we plan to extend our study by investigating further on how to not miss any results and still outperform RDF-3X. We plan to give high weights to important edges and see how that helps us in partitioning and querying. Later, we also plan to try other graph partitioning technique called hmetis. For hmetis, we need to identify and define hyperedges for a RDF graph, which is a challenge. A hyperedge is a collection of many edges and is similar to a small graph.

APPENDIX A

ALGORITHMS

Algorithm 1 : Convert to IDs

```
Proc ConvertToIDs()
Let V be the sorted list of distinct subjects, predicates and objects.
Let M be a map that stores (key, value) pairs where all keys are distinct
Let Y be the dataset file
i ← 1
Foreach item in V
        M ← (item, i)
        i ← i + 1
end
Foreach (s,p,o) in Y
        Write_to_file (M[s], M[p], M[o])
        Write_to_file "\n"
End
```

Algorithm 2: Generate gpmetis input file from the dataset (Type – I)

Proc GenerateGpmetisInputFormattedFile()

Let V be the sorted list of subjects and objects.

Let M be map with stores (key, value) pairs where all keys are distinct

Let D be the dataset with all triples id-formatted n-triples.

Let A_i be the list of IDs that contains the connected vertices (subjects/objects) of vertex i

V_c ← count of vertices

E_c ← count of edges (number of triples in D)

Foreach vertex i in V

A_i ← "a_1 a_2 a_3 . . . a_k" where 0 <= k <= V_c and {a_1, a_2, a_3, . . . a_k} are connected vertices to vertex i in Dataset D

$Value_i$ ← A_i

M ← (i, $Value_i$)

End

Write_ to_file "V_c E_c"

For i ← 1 to V_c

Write_to_file M[i]

Write_to_file "\n"

End

Algorithm 3: Generate gpmetis input file from the dataset (Type – II)

Proc GenerateGpmetisInputFormattedFile()

Let V be the sorted list of subjects and objects.

Let M be map with stores (key, value) pairs where all keys are distinct

Let D be the dataset with all triples id-formatted n-triples.

Let A_i be the list of IDs that contains the connected vertices (subjects/objects) of vertex i

Let W be the list of weights of vertices. W_i is the weight of vertex i.

$V_c \leftarrow |V|$ (count of vertices)

$E_c \leftarrow |E|$ (count of edges i.e., number of triples in D)

Foreach vertex i in V

$A_i \leftarrow$ "$a_1\ a_2\ a_3\ \ldots\ a_k$" where $0 <= k <= V_c$ and $\{a_1, a_2, a_3, \ldots a_k\}$ are connected vertices to vertex i in Dataset D

$Value_i \leftarrow W_i$ + " " + A_i

$M \leftarrow (i, Value_i)$

End

Write_ to_file "$V_c\ E_c$"

For i $\leftarrow$ 1 to V_c

Write_to_file M(i)

Write_to_file "\n"

End

Algorithm 4: Generate gpmetis input file from the dataset (Type – III)

Proc GenerateGpmetisInputFormattedFile()

Let V be the sorted list of subjects and objects.

Let M be map with stores (key, value) pairs where all keys are distinct

Let D be the dataset with all triples id-formatted n-triples.

Let A_i be the list of IDs that contains the connected vertices (subjects/objects) of vertex i

Let E be the list of weights of edges. e_{ij} is the weight of edge (i,j).

$V_c \leftarrow |V|$ i.e., count of vertices

$E_c \leftarrow |E|$ i.e., count of edges i.e., number of triples in D

Foreach vertex i in V

$A_i \leftarrow$ "$a_1\ e_{i1}\ a_2\ e_{i2}\ a_3\ e_{i3}\ \ldots\ a_k\ e_{ik}$" where $0 <= k <= V_c$; $\{a_1, a_2, a_3, \ldots a_k\}$ are connected vertices to vertex i in Dataset D

$Value_i \leftarrow A_i$

$M \leftarrow (i, Value_i)$

End

Write_ to_file "$V_c\ E_c$"

For i $\leftarrow$ 1 to V_c

Write_to_file M[i]

Write_to_file "\n"

End

Algorithm 5: Generate gpmetis input file from the dataset (Type – IV)

Proc GenerateGpmetisInputFormattedFile()

Let V be the sorted list of subjects and objects.

Let M be map with stores (key, value) pairs where all keys are distinct

Let D be the dataset with all triples id-formatted n-triples.

Let A_i be the list of IDs that contains the connected vertices (subjects/objects) of vertex i

Let E be the list of weights of edges. e_{ij} is the weight of edge (i,j).

Let W be the list of weights of vertices. W_i is the weight of vertex i.

$V_c \leftarrow |V|$ i.e., count of vertices

$E_c \leftarrow |E|$ i.e., count of edges i.e., number of triples in D

Foreach vertex i in V

$A_i \leftarrow$ "$a_1\ e_{i1}\ a_2\ e_{i2}\ a_3\ e_{i3}\ \ldots\ a_k\ e_{ik}$" where $0 <= k <= V_c$; $\{a_1, a_2, a_3, \ldots a_k\}$ are connected vertices to vertex i in Dataset D

$Value_i \leftarrow W_i$ + " " + A_i

$M \leftarrow (i, Value_i)$

End

Write_ to_file "$V_c\ E_c$"

For i $\leftarrow$ 1 to V_c

Write_to_file M(i)

Write_to_file "\n"

End

Algorithm 6: Get triples in each partition

Proc GetGraphInEachPartition()

Let n be the number of partitions

Let O be map with stores (key, value) pairs where a key is any vertexid and the corresponding value is the partitionNumber where it is put by gpmetis.

A_p ←list of all vertices from O where value/partitionnumber = p

/*loop for all partitionids*/

Foreach p = 0 to n-1

    Foreach key k in O

        if(O[k] == p)

            A_p ← add k to the list

        end if

    end

    /* loop for each vertex in A_p */

    Foreach vertex V in A_p

        $Partition_p$ ← ntriples originating from/to V

    End

    /*Remove duplicate triples in $Partition_p$ */

    Sort($Partition_p$) /* unix shell command*/

    Unique($Partition_p$) /* unix shell command */

End

Algorithm 7: Get list of edge cuts as a result of graph partitioning

```
Proc GetEdgeCuts()
Let n be the number of partitions
Let O be map with stores (key, value) pairs where a key is any vertexid and the
corresponding value is the partitionNumber where it is put by gpmetis.
Let Y be the whole dataset in ntriples format (subject, predicate, object)
Foreach (s, p, o) in Y
        if(O[s] != O[o]) then
                write_to_file “(s, p, o)”
                write_to_file “\n”
        end if
end
```

Algorithm 8: Convert IDs to string formatted ntriples

```
Proc ConvertIDToTriples()
Let V be the sorted list of distinct subjects, predicates and objects.
Let M be a map that stores (key, value) pairs where all keys are distinct
Let Y be the dataset file
i ← 1
Foreach item in V
        M ← (i, item)
        i ← i + 1
end
Foreach (s,p,o) in Y
        Write_to_file (M[s], M[p], M[o])
        Write_to_file "\n"
End
```

REFERENCES

1. Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411-422, 2007.
2. Soa Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In *SemWeb*, 2001.
3. Kemafor Anyanwu, Angela Maduko, and Amit P. Sheth. SPARQ2L: towards support for subgraph extraction queries in rdf databases. In *WWW*, pages 797-806, 2007.
4. Liu Baolin and Hu Bo. HPRD: A high performance RDF database. In *NPC*, pages 364-374, 2007.
5. Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: An architecture for storing and querying RDF data and schema information. In *Spinning the Semantic Web*, pages 197-222, 2003.
6. C-Store. http://db.csail.mit.edu/projects/cstore/
7. Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *VLDB*, pages 1-10, 2000
8. Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB*, pages 1216-1227, 2005.

9. Andreas Harth, Jurgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A federated repository for querying graph structured data from the web. In *ISWC/ASWC*, pages 211-224, 2007.

10. Olaf Hartig and Ralf Heese. The SPARQL query graph model for query optimization. In *ESWC*, pages 564-578, 2007.

11. Jena: a Semantic Web Framework for Java. http://jena.sourceforge.net/.

12. Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647{659, 2008.

13. Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.,* 33(4):351-385, 1996.

14. Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697-706, 2007.

15. Yannis Theoharis, Vassilis Christophides, and Gregory Karvounarakis. Benchmarking database representations of rdf/s stores. In *International Semantic Web Conference*, pages 685-701, 2005.

16. Octavian Udrea, Andrea Pugliese, and V. S. Subrahmanian. GRIN: A graph based RDF index. In *AAAI*, pages 1465-1470, 2007.

17. Uniprot RDF. http:/dev.isb-sib.ch/projects/uniprot-rdf/.

18. Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008-1019, 2008.

19. Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in Jena2. In *SWDB*, pages 131-150, 2003.

20. *W3C*: RDF/OWL representation of WordNet. http://www.w3.org/TR/wordnet-rdf/.

21. Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In Proceedings of the 19th international conference on World Wide Web, pages 41–50, New York, NY, 2010.

22. Feida Zhu, Xifeng Yan, Jiawei Han, and Philip S. Yu. gPrune: A constraint pushing framework for graph pattern mining. In *PAKDD*, pages 388-400, 2007.

23. Hyunsik Choi, Jihoon Son, YongHyun Cho, Min Kyoung Sung, and Yon Dohn Chung. SPIDER: a system for scalable, parallel / distributed evaluation of large-scale RDF data. In CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management, pages 2087–2088, New York, NY, 2009.

24. Stephen Harris and Nicholas Gibbins. 3store: Efficient Bulk RDF Storage. *In Practical and Scalable Semantic Systems*, 2003.

25. Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.

26. Mohammad Farhan Husain, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools. *Cloud Computing, IEEE International Conference* on, pages 1–10, 2010.

27. Maciej Janik and Krys Kochut. BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In

Proceedings of the 4th International Semantic Web Conference, pages 431–445, 2005.

28. George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20:359–392, December 1998.

29. Youn Hee Kim, Byung Gon Kim, Jaeho Lee, and Hae Chull Lim. The path index for query processing on RDF and RDF schema. In *Advanced Communication Technology, 2005, ICACT 2005. The 7th International Conference on*, volume 2, pages 1237–1240, 2005.

30. Justin J. Levandoski and Mohamed F. Mokbel. RDF Data-Centric Storage. In *Proceedings of the 2009 IEEE International Conference on Web Services*, pages 911–918, Washington, DC, 2009.

31. Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2–3):158–182, 2005.

32. Olaf Hartig and Ralf Heese. The SPARQL Query Graph Model for Query Optimization. pages 564–578. 2007.

33. Oktie Hassanzadeh, Anastasios Kementsietsidis, and Yannis Velegrakis. Data management issues on the semantic web. In *ICDE*, pages 1204–1206, 2012.

34. Resource Description Framework (RDF). http://www.w3.org/RDF/.

35. Bet Buy jump starts data web marketing. http://www.chiefmartec.com/2009/12/best-buy-jump-starts-data-web-marketing.html.

36. 4store - scalable RDF storage. http://4store.org/.

37. DBpedia dataset. http://dbpedia.org/.

38. Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. The VLDB Journal, 19(1):91–113, Feb. 2010.

39. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

40. The New York Times Linked Open Data. http://data.nytimes.com/.

41. JDBM2 Package. http://jdbm2.googlecode.com/svn/trunk/javadoc/index.html?jdbm/package-summary.html

42.

VITA

Dinesh Barenkala was born on September 25, 1987 in Andhra Pradesh, India. He received a Bachelor's degree from Osmania University, Hyderabad in 2009 and worked as a Systems Engineer in Infosys Ltd until 2010. In December 2010, he came to the United States to pursue Master's degree in Computer Sciences at the University of Missouri – Kansas City (UMKC). He worked as a part-time Web Developer at USSourceLink for 6 months and then moved on to Research under Dr. Praveen Rao. Currently, he is working as a Sr Software Developer at DST Systems, Kansas City.