RGIS: EFFICIENT REPRESENTATION, INDEXING AND QUERYING OF LARGE RDF GRAPHS

by

JAGALPURE, ANIRUDDHA GIRISH

(Under the Direction of Lakshmish Ramaswamy)

ABSTRACT

RDF data is a labeled directed graph. SPARQL is an RDF Query language that is used to extract information from the RDF Graph. There are different RDF Engines like Sesame, RDF-3X, OWLIM & Jena. Jena is the most popular framework and is widely used. Jena In-Memory model cannot scale for large RDF datasets while Jena SDB and Jena TDB have high latencies. In this thesis we propose a new system *'RGIS' (RDF Graph Split and Index)* for processing SPARQL queries on RDF data. *RGIS* is not only scalable but also faster than Jena and OWLIM-SE (BigOWLIM). *RGIS* uses a custom data format and novel indexing technique to store the RDF data. Our custom format stores the RDF data into different files based on Classes and Object Properties present in the RDF data. These files are then given an index and each instance in these files is given a unique index value. We have also developed an RDF structure-aware Query Planner that uses the topology of RDF graph to intelligently schedule various query operations. When compared with Jena TDB, OWLIM and Mulgara on LUBM datasets, *RGIS* was not only had faster response times but also has less memory overhead.

INDEX WORDS: RDF, SPARQL, Query Processing, Graph, Jena and Hadoop.

RGIS: EFFICIENT REPRESENTSTION, INDEXING AND QUERYING OF LARGE RDF GRAPHS

by

JAGALPURE, ANIRUDDHA GIRISH

B.E., University of Pune, India, 2008

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2012

RGIS: EFFICIENT REPRESENTSTION, INDEXING AND QUERYING OF LARGE RDF GRAPHS

by

JAGALPURE, ANIRUDDHA GIRISH

Major Professor:	Lakshmish Ramaswamy
Committee:	John A. Miller
	Ismailcem Budak Arpinar

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2012

DEDICATION

To my parents, family & friends, for their love, support and encouragement.

ACKNOWLEDGEMENTS

Past 2.5 years are the most enriching years of my life and has been a big learning curve. I take this opportunity to thank Dr. Lakshmish Ramaswamy for his constant support, feedback, encouragement and motivation. I would like to thank Dr. John A. Miller for his constant inputs, guidance and motivation for my project. I would also like to thank Dr. Ismailcem Budak Arpinar for providing his valuable inputs and motivation for my project.

I would also like to thank my friends Akshay, Chinmay & Siva for all the support and motivation. A special thanks to Kat Gilmore and Sue Myers Smith for their love, support, motivation and giving me an opportunity to work as lead web developer for the UGA College of Veterinary Medicine.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

Currently, the web pages are dominated by unstructured and semi-structured format of data on the web pages. Besides, humans are able to process the information that is available on the web. Since this information is unstructured, machines cannot interpret the information that is presented on the web. Hence, with an objective of converting this unstructured data into structured data and make it more machine readable, The W3C Consortium proposed a standard Semantic Web. Semantic Web promotes a common data format on the web. Thus, Semantic Web provides a common framework for data to be shared across the web, platforms and different enterprises. To accomplish the objectives of Semantic Web, in 1999 The W3C proposed the Resource Description Framework (RDF), a meta-data data model used to exchange data on the Web. After its introduction, it has been popular in Semantic Web Technologies and Life Sciences. Besides, [8] major search engines like Google (RichSnippets) and Yahoo (SearchMonkey) have started displaying webpages marked up with RDF more prominently in search results and hence encouraging usage of RDF.

Data in RDF is represented in the form of Triples of Subject, Predicate and Object; and each of them has a Uniform Resource Identifier (URI). Hence, a group of RDF statements can then form a labeled directed graph. In such a graph, the nodes (Subject and Object) are two URI's that are linked by an edge (Predicate) and this edge also has its own URI. Thus, RDF is a directed, labeled graph data format that is used to represent information.

E.g. Department0 of University0 can be represented in RDF format as shown below:

```
<ub:Department rdf:about="http://www.Department0.University0.edu">
    <ub:name>Department0</ub:name>
    <ub:subOrganizationOf>
        <ub:University rdf:about="http://www.University0.edu" />
    </ub:subOrganizationOf>
</ub:Department>
```

As seen in above example, information is represented in a complex structure which is difficult to understand for humans. Hence, there is a need for a format in which RDF data can be represented in human-readable format. Hence, W3C proposed a new standard Notation 3 [14][15], the compact representation of RDF's XML data, which is much simpler and human readable format than RDF. Turtle (Terse RDF Triple Language)[31], is a serialization format for RDF data model. It is a subset of Notation3.

E.g. Department0 of University0 can be represented in N3 format as shown below:

```
<http://www.Department0.University0.edu> a ub:Department ;
                                         ub:name "Department0" ;
                                         ub:subOrganizationOf
<http://www.University0.edu> .
```

Thus, as seen above, the information is much easier to read and for human consumption.

Since RDF is a directed, labeled graph format to represent data, there also needs to be a standard that needs to be established to retrieve the information from the RDF dataset. Hence to retrieve information from RDF graphs, W3C proposed a new Query Language called SPARQL. SPARQL (SPARQL Protocol and RDF Query Language)[18][19] is an RDF query language that is used to retrieve and manipulate the data that is stored in RDF format. It provides the standards and keywords that need to be used to form a Query to retrieve information from an RDF dataset. Below is an example, represented in the SPARQL query Language, to retrieve all *Graduate Students* taking course *GraduateCourse0*.

E.g.

```
SELECT ?X
WHERE {
        ?X rdf:type ub:GraduateStudent .
        ?X ub:takesCourse
<http://www.Department0.University0.edu/GraduateCourse0>}
```

1.1 MOTIVATION

RDF engines support querying, storing & indexing of RDF data. There are many RDF engines like Jena [27], OWLIM [28], Sesame [29] and RDF-3X [30]. Among all these, Jena is the most popular framework. In Jena, to store the data, there are several models namely:

- Jena In-Memory:
- Jena SDB
- Jena TDB

Previous research [1] [4] has shown that since Jena In-Memory model loads all the RDF data into main memory, Jena In-Memory model cannot load for more than 10 Million triples for 2GB memory. Hence, for larger data sets, it is not possible to use Jena In-Memory Model. Jena SDB can load up to 650Million Triples and Jena TDB can load up to 1.7 Billion Triples. However

previous research [23][24] has shown that these models have high latencies. Hence, in terms of performance, both these systems are slow. Another important drawback with all these models is that they are meant to work on a single machine. Hence, there is a need to develop a system, which can scale to support big data sets and at the same time is efficient and takes less memory and time to evaluate the queries. Besides, the system should also be implemented on a single machine and should also work on a distributed framework like Hadoop.

1.2 CONTRIBUTIONS

As there is a need to develop scalable, memory efficient and performance efficient system for RDF data, we have developed a new system **RGIS (RDF Graph Split and Index)** which uses its own custom format and indexing structure to store the RDF data and uses custom Query Plan Generator which is used to extract information from the custom data format used by the system. Hence, we have two most important contributions to make:

- Custom Data format and Indexing structure:

  We create a separate file for each Class and Property in the RDF data and give a unique index to each of the files and items in these files. All the instances of a class are then stored in the respective Class File and each instance is given a unique index. Data in Property Files is stored in two sections: Header and Body. In Header, we store all the Meta-data about the Subject and Objects. Whereas in Body, we store the Subject and Object that are linked to each other by the Predicate that is the name of the Property File. Instead of storing the URI, we used the index that has been assigned to the Subject/Object in their respective Class Files.

- Query Plan Generator:

  The Query Plan Generator provides us with a query plan to execute the SPARQL query asked by the user. The Query Plan Generator examines the query and gives us the order in which the triples need to be executed. Since, we are storing the information in our Custom Format, it also provides information to which files needs to be loaded into the memory to execute the query.

- Query Plan Executor:

  Once the query plan is received from the Query Plan Executor, the Query Plan Executor loads the required files in the memory and executes the order of triples it has received in the Query Plan.

Thus, by using our custom data format, we are able to reduce the RDF file size by up to 66%. By using indexes and selectively loading the data files into the memory, RGIS uses less memory to perform complex operations and is faster than Jena TDB and OWLIM-SE.

1.3 ORGANIZATION

In Chapter 2, we discuss the System Architecture of RGIS. We would discuss the different components that are involved in RGIS and their interaction with each other.

In Chapter 3, we discuss the new custom storage and indexing format to store the RDF data. We would also explain the steps involved to convert he RDF data into the new custom format.

In Chapter 4, we discuss the algorithm the Query Plan Generator uses to generate the Query Plan.

In Chapter 5, we discuss the details of the execution of Query Plan Executor. The Query Plan Executor is responsible to execute the Query Plan.

In Chapter 6, we discuss our experimental setup to test the working of RGIS. We would also discuss the performance (time and memory utilization) of the system and compare it with Jena TDB.

In Chapter 7, we discuss the previous research and related work. We would also discuss on how RGIS differentiates from the other existing systems.

In Chapter 8, we discuss about our conclusions and the future work that we are working on.

CHAPTER 2

SYSTEM ARCHIECTURE

RGIS comprises of three major components: Pre-processor, Query Plan Generator and Query Plan Executor. As discussed in previous section, RGIS stores RDF data in its custom indexed format. Hence, the Pre-Processor pre-processes the RDF data and converts into the custom indexed format. The Query Plan Generator takes the SPARQL query as an input from the user and generates the Query Plan that needs to be executed. The Query Plan consists of the sequence of steps be performed and the files that needs to be loaded into the memory to evaluate the SPARQL query. The Query Plan is then forwarded to Query Plan Executor that then loads the pre-processed files and then executes the steps in the Query Plan. Below is the diagrammatic representation of RGIS.

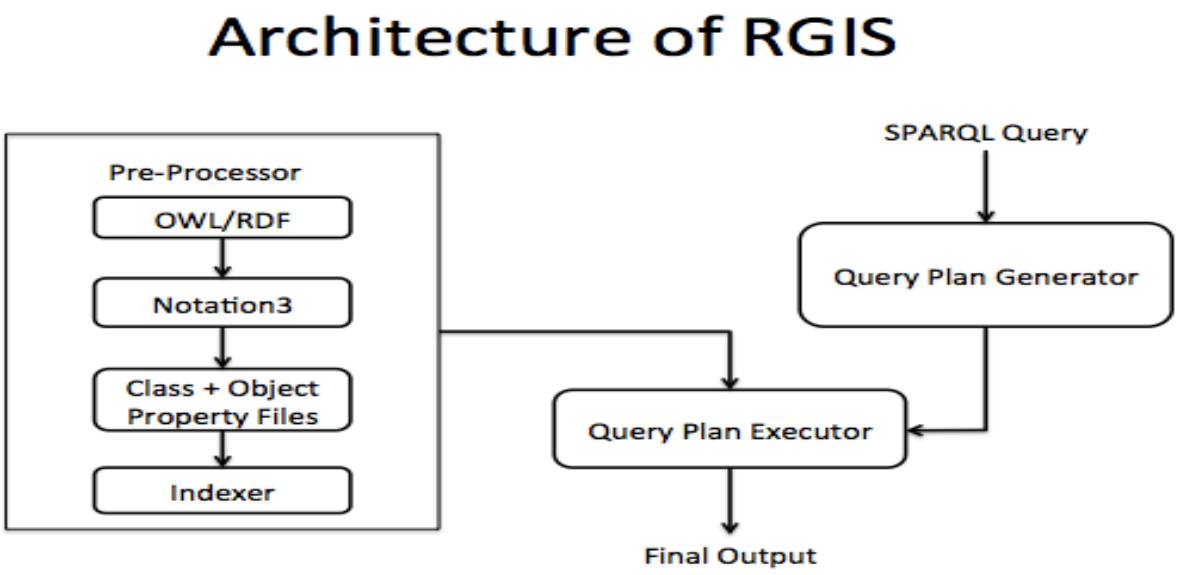

Figure 1: Architecture of RGIS

Now, lets discuss in detail the individual working on each of system components mentioned above:

1. Pre-Processor:

   RGIS uses its Custom Format to store the RDF data. In RGIS, RDF data is stored in *'Class Files'* and *'Property Files'*.

   - Class Files:

   For every class in the RDF data, we create a *Class File*, which is named after that class. All instances of a class are stored in the respective '*Class File'* of its class.

   Now, there can be hierarchy if class in the RDF data. We store this hierarchical information in a special file called as 'Class Hierarchy'.

   - Property Files:

   For every property found in the ontology, we create an *Property File*, which is named after the property. Thus, we store the subject and the object, which are linked by a given property in the respective *Property File*.

   Hence, the primary task of the preprocessor is to convert the RDF data into Class Files and Property Files. RGIS uses its custom indexing technique. Hence, once the RDF data is converted into Class Files and Property Files, the pre-processor indexes these files and also the data inside the files. The details of indexing would be discussed in section 3.2.

   Further details of preprocessor are discussed in Chapter 3.

2. Query Plan Generator:

   The Query Plan Generator takes the SPARQL query as an input and produces a query execution plan that is then forwarded to the Query Plan Executor to execute it. The Query Plan Generator examines the user defined SPARQL query and analyzes it. It then gives

us the order in which triples needs to evaluate and also the Meta-data information on which files needs to be loaded into the memory to evaluate the query. The details of how a Query Plan is generated are discussed in Chapter 4.

3. Query Plan Executor:

   The Query Plan Executor executes the Query Plan that is generated by the Query Plan Generator. It loads the files mentioned in the Meta-data information of the Query Plan. Hence, RGIS selectively loads the data into memory. Once the data is loaded into memory, it executes the Query Plan to generate the results that are then displayed to the user. The details of execution of Query Plan Executor are discussed in Chapter 5.

CHAPTER 3

INDEXING CLASS FILES AND PROPERTY FILES

RGIS uses its custom index data format to store the RDF data into Class Files and Property Files. Hence, the RDF data needs to be preprocessed and converted into the required format. In this chapter, we would discuss in details about how to convert the RDF data into the custom index data format used by RGIS.

3.1 RDF TO NOTATION3 FORMAT:

As seen in the figure1 above, the first step in the preprocessing of the data is converting the RDF data into Notation3 format. These Notation3 files are then provided as input files for the next step discussed below.

3.2 CLASS FILES & PROPERTY FILES:

Now, in this sub-module, we first find all the *Classes* and *Object Properties* in the ontology and create a separate file for each of them. Now, the Notation3 files obtained in the above step act as input files here. The Notation3 files are read and the data in these files is reorganized based upon the *Classes* and *Object Properties*. Thus, depending upon the *Class* or *Property* that is read from the Notation3 file, data is then added into respective *Class File* or *Property File*.

E.g.:

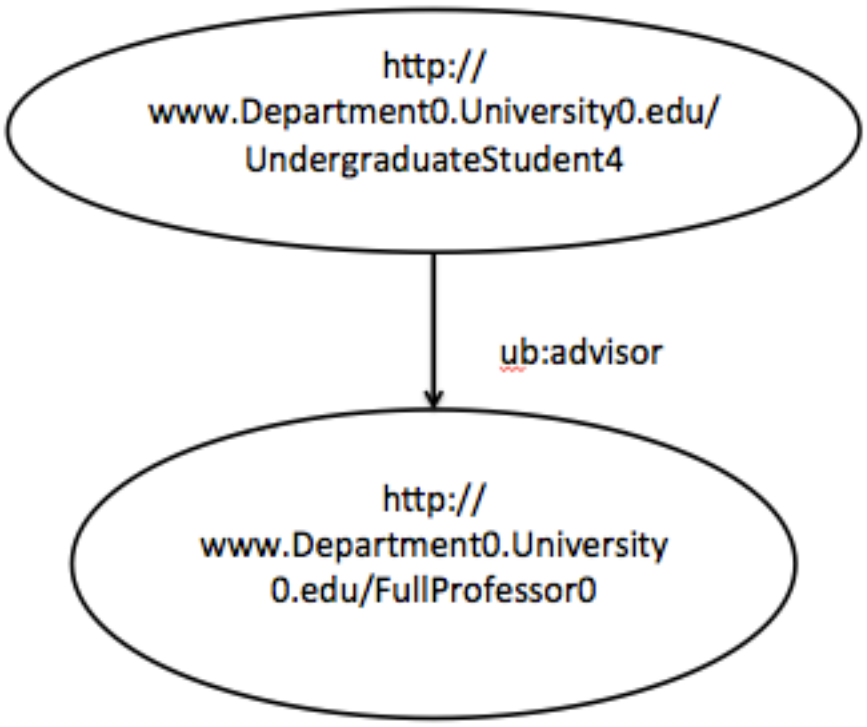

Figure 2: Subject-Predicate-Object Example

As seen in the above figure, we have:

- Classes: Student, Professor.
- Property: advisor
- Data (Instances): FullProfessor0, UndergraduateStudent4

Hence, when this information is provided to the pre-processor, the preprocessor would create 3 files: *Student, Professor & advisor*. Hence, the above information will be represented as following in these 3 files:

Class Files:

Student	Professor
UndergraduateStudent4	FullProfessor0

Table 1: Example - Class File

Property Files:

advisor	
UndergraduateStudent4	FullProfessor0

Table 2: Example - Property File

3.3 INDEXING CLASS FILES & PROPERTY:

Now, in this step, the *Class Files* and *Property Files* obtained above are each given a unique numeric value called as *'Class File Index'*.

- Indexing Class Files:

  Each of the class files is given a unique numeric value called as *'Class File Index'*. Hence, consider the example that we discussed in section 3.1.2, the class files, *Students* & *Professor* are given index *'1'* & *'2'* respectively and stored as: *'1.Student'* & *'2.Professor'*.

  Now, the data inside these class files are nothing but instances of the class that were found in the OWL/RDF files that we processed earlier. Each of these instances inside every class file is given a numeric value, which we call it as *'Class Instance Index'*. The format of *'Class Instance Index'* is:

  Class Instance Index = ClassIndex.UniqueNumericValue

Now, the data inside these class files is rearranged and stored in the format shown:

ClassIndex.ClassName	
Class Instance	ClassInstanceIndex

Table 3: Indexed Class File

E.g.:

1.Student	
UndergraduateStudent4	1.1
GraduateStudent4	1.2

Table 4: Example - Indexed Class File

2.Professor	
FullProfessor0	2.1
FullProfessor8	2.2

Table 5: Example – Indexed Class File

- Indexing Property Files:

  The information in the *Indexed Object Files* is stored in two sections in the file: *Header* and *Body*, which would be discussed below.

  Body:

  Similar to Class Files, Property Files are also given a unique index, which we would call as *'Property File Index'*. Now, each row in this file represents a Subject-Object value pair, which is linked by the Property represented by the Property File's name. Thus, these instances are replaced by their respective *'Class Instance Index'* discussed above. Thus, this information is stored in the 'Body' section of the Object Indexed File.

  Header:

Further, we also store information about the 'Class Index' for subject and object that are present in a given Property File. This information is stored in the 'Header' section of the file.

E.g.: Consider the example discussed above. The data for Property File *'advisor'* would be converted into:

3.advisor	
Header:	
Subject: 1	
Object: 2	
Body:	
1.1	2.1
1.7	2.5

Table 6: Example – Indexed Property File

Thus, by using a numeric value instead of the whole URI to represent an instance of a Class, we were able to reduce the size of the files considerably. The details of it would be provided in coming sections.

CHAPTER 4

QUERY PLAN GENERATOR

There are two objectives for the Query Plan Generator:

- Determine the order in which we need to execute the query
- As data is stored in different files, we need to determine from which files we need to pull the data.

Let us see in details how these two objectives are met.

Determining the order of execution of the Query is important because the information requested by the user might be dependent on each other. Hence, we need to determine a path that does not involve any inter-dependencies. To accomplish this we follow the steps below:

Pseudo Code:

Nodes ← getAllVariables (SPARQL Query)

For i: 1 to |Nodes|

    Nodes[i] ← getClassProperties()

Triples ← getTriples(SPARQL Query)

For i: 1 to |Triples|

    If Subject is variable

        Nodes[Subject].ObjectPropertyList ← Triple [i]

    Else

        Nodes[Object].ObjectPropertyList ← Triple [i]

While Nodes != empty

N = getIndependentNode(Nodes)

OutputList ← N

Nodes.remove(N)

Explanation of Algorithm:

1. We determine all the variables that are present in the query. Each variable is considered as a node.
2. We then compare these variables with the variables that are required in the output and determine which of these nodes would be our output nodes.
3. We then determine the 'type' i.e., 'Class' of the node. The Class information can be found from two places:
    a. The class of the node can be specified in the Query

       E.g.: *?X ub:type Department*
    b. From the 'Header' section of the Object Indexed File.
4. Once we have determined the Class information about a node, we then add the Object properties that are asked in the query to the respective nodes. Thus, every node has its own *'Property List'*, which contains the Object Properties.
5. After adding the Object Properties, we check every node for any dependencies and for every node we create a dependency list.
6. Then, we select a node, which does not have any dependency and add it to the Output List.
7. Then, we remove this node, which we added in the Output List, from the dependency list of all other nodes.

8. We repeat steps 6 & 7 till all nodes are added into the Output List.

Thus, the order in which the nodes are added into the Output List gives us the order in which we need to execute the query plan.

E.g.: Consider the following query:

SELECT ?X, ?Y, ?Z

WHERE

{ ?X rdf:type ub:GraduateStudent .

?Y rdf:type ub:University .

?Z rdf:type ub:Department .

?X ub:memberOf ?Z .

?Z ub:subOrganizationOf ?Y .

?X ub:undergraduateDegreeFrom ?Y}

For the query above, we would generate a following graph for Steps 1-5:

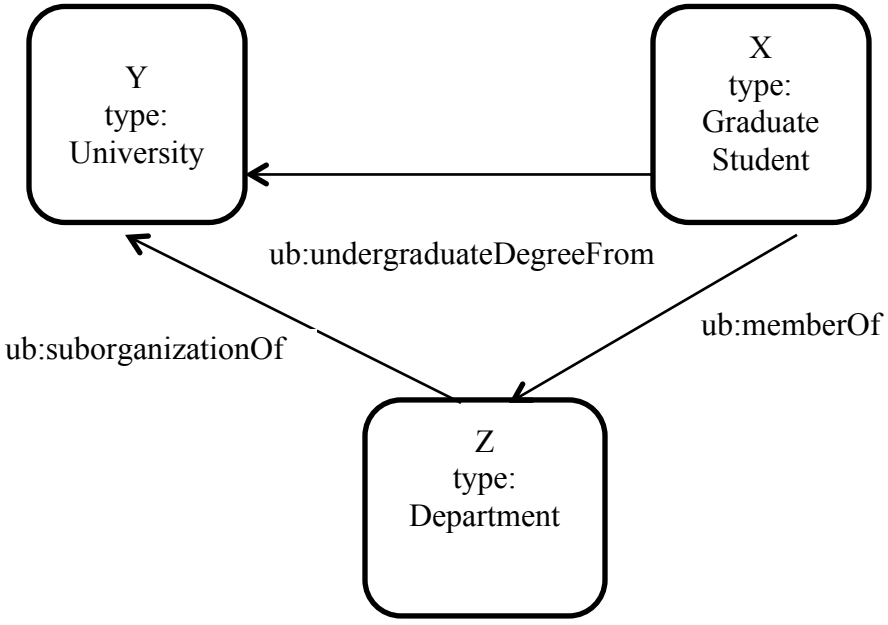

Figure 3: Graph Generated for SPARQL Query

Thus, if we follow Steps 1-5 discussed above, we get three nodes: X, Y & Z. The dependencies of each node can be then express as shown below:

Node	Dependency on Node	Property List
X	Y,Z	type: GraduateStudent underGraduateDegreeFrom: X memberOf: Z
Y	-	type: University
Z	X	type: Department suborganizationOf: Y

Table 7: Node Dependency List

Hence, if we follow Steps 6-8 discussed in this section, then the order of execution of nodes would be as follows:

Node	Property List
Y	type: University
Z	type: Department suborganizationOf: Y
X	type: GraduateStudent underGraduateDegreeFrom: X memberOf: Z

Table 8: Property List of Nodes

Now, once we have determined the order of execution, we now need to determine the files that are required to evaluate the query. As discussed in Chapter 3, we store all the RDF data in Class Files and Property Files.

In the Query Plan, as seen in above example, we also generate the Property List. Hence, the entries in the Property List would determine which files needs to be used and loaded to evaluate the query. Hence, for the above example, we would be loading the following files:

Class Files:

- University
- Department
- GraduateStudent

Property Files:

- suborganizationOf
- undergraduateDegreeFrom
- memberOf

CHAPTER 5

QUERY PLAN EXECUTOR

The Query Plan generated by Query Plan Generator is then forwarded to the Query Plan executor. The main objective of Query Plan executor is to execute the Query Plan and give the required output to the user.

Below are the steps that are followed by the Query Plan Executor to execute the Query Plan:

1. Select the node from the Query Plan.
2. Check that in the Object properties of the selected node are there any literals.
    a. If literals are found, then place the Object Properties with literals at the start of the *Property List*. By evaluating literals first, we can reduce the search space for the query we are evaluating.
    b. If literals are not found, do not change the *Property List*.
3. Now, for the selected node, evaluate the Property List.
    a. For first property in the Property List:
        i. Go to the respective Property File, which is loaded in the memory.
        ii. Get the required objects from the file and add it to the output list of that respective node.
    b. For rest of Properties in the List:
        i. Go to the respective Property File, which is loaded in the memory.

ii. Use the Output List obtained in above step and use it as an input to get the required information from the loaded Property File. Since the Property File is stored as a Hash Map, the retrieval of the objects is quick.

iii. Add the new relevant objects to the Output List.

iv. Repeat the above steps for all the Object Properties in the List.

CHAPTER 6

RESULTS

6.1 EXPERIMENTSL SETUP:

To test the functionality of RGIS, we have used the Leigh University Benchmark, popularly known as LUBM [25]. LUBM is one of the most popular benchmark for the evaluation of Semantic Web Repositories. This benchmark is extensively used to measure the performance of the repository when querying on large data sets. This benchmark contains:

- Ontology with University as its domain
- Customizable and repeatable synthetic data
- 14 test queries [26]
- Performance Metrics

We have used Data Generator (UBA) provided by LUBM to generate our dataset. Thus, to test RGIS, we generated two data sets and the details are as follows

Data Set	Size of OWL/RDF Files	No of Triples (In Millions)	Size of files after Indexing	Disk Space savings after Indexing
Data Set 1	3.42 GB	41.27	1.14 GB	66.66 %
Data Set 2	6.90 GB	82.92	2.34 GB	66.08%

Table 9: Data Sets used for Test

Thus, as seen from the table above, when the RDF data is processed and converted and stored in our custom format, we were able to reduce the size of the files by 66%.

To test RGIS, we used a machine having 2.5GHz Intel Core i5 processor, 8GB of RAM, Mac OS X 10.8.2 and Java 1.6. For testing the response time of RGIS on the LUBM test queries, we used different Java heap space memory settings.

6.2 PERFORMANCE OF RGIS FOR DIFFERENT HEAP SIZE:

In this section, we will compare the performance of RGIS on the two datasets and on different memory heap sizes of JVM. This is an important evaluation of the system as this will give us insight on the response time of RGIS to answer a query and also give us information about the memory utilization to answer the query. Thus, from the results we can determine the minimum size of the heap that is required to answer the queries.

6.2.1 HEAP SIZE: 4GB

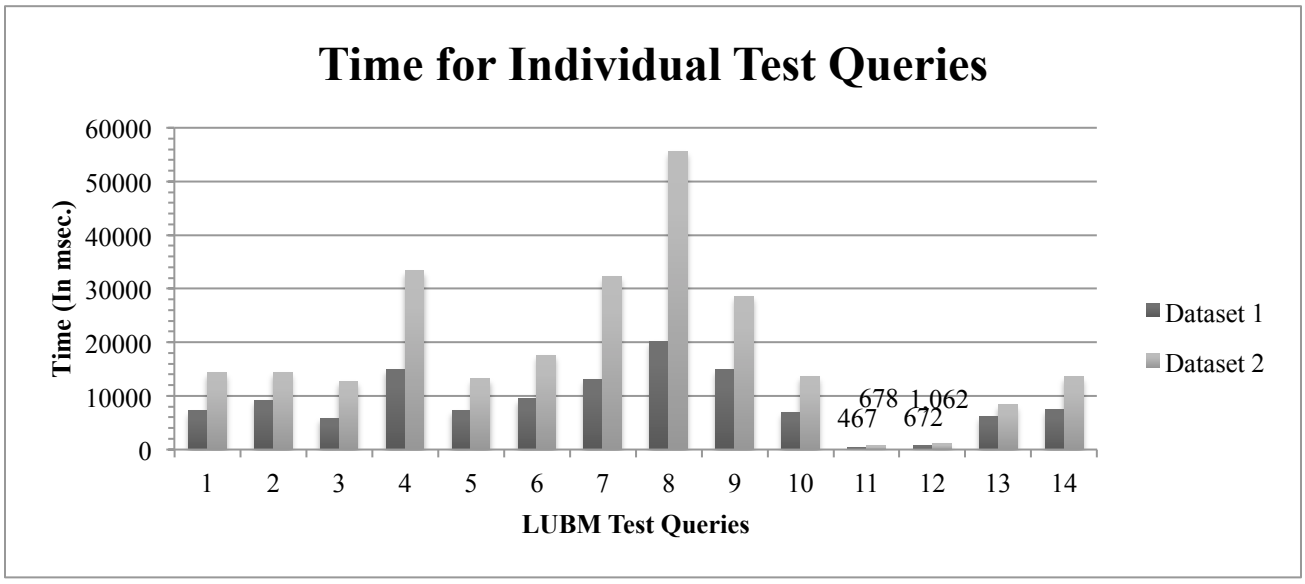

Figure 4: Time for Test Queries v/s 4GB Memory Space

Above figure shows the response time of the system to answer query for both the data sets when the heap size of JVM is set to 4GB.

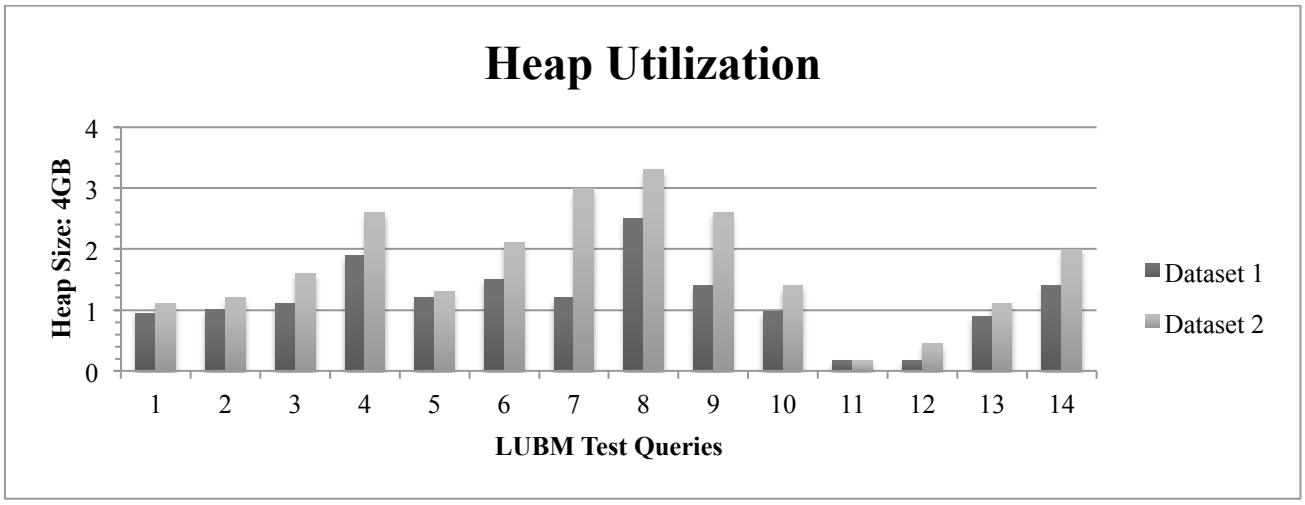

Figure 5: Memory Utilization for 4GB

Above figure shows the Heap utilization of the system to answer query for both the data sets when the heap size of JVM is set to 4GB.

6.2.2 HEAP SIZE: 3GB

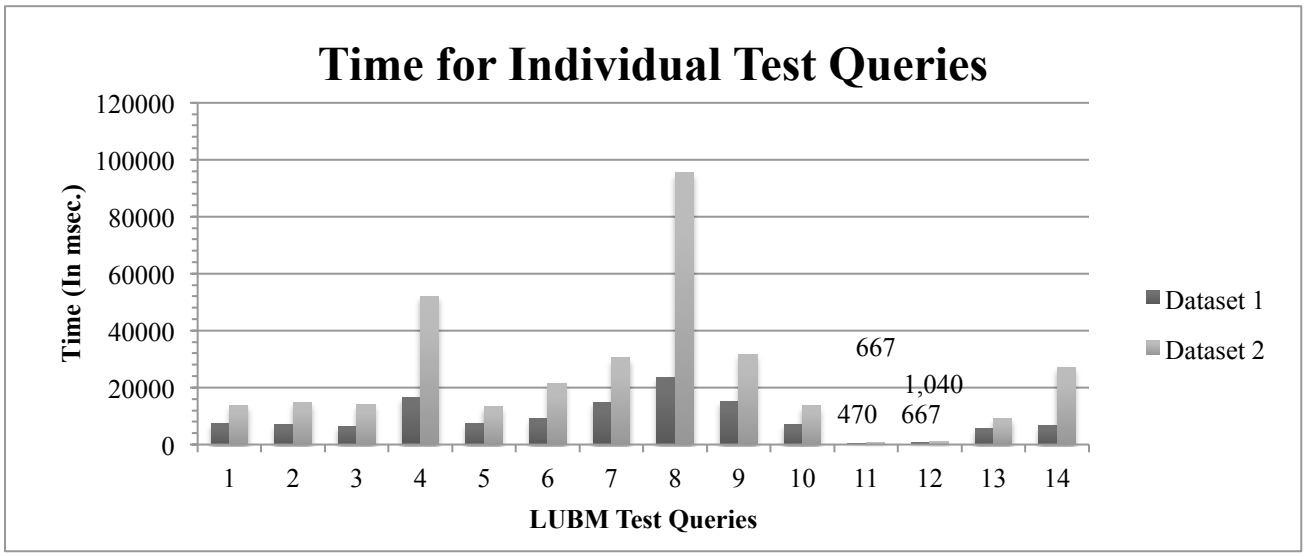

Figure 6: Time for Test Queries v/s 3GB Memory Space

Above figure shows the response time of the system to answer query for both the data sets when the heap size of JVM is set to 3GB.

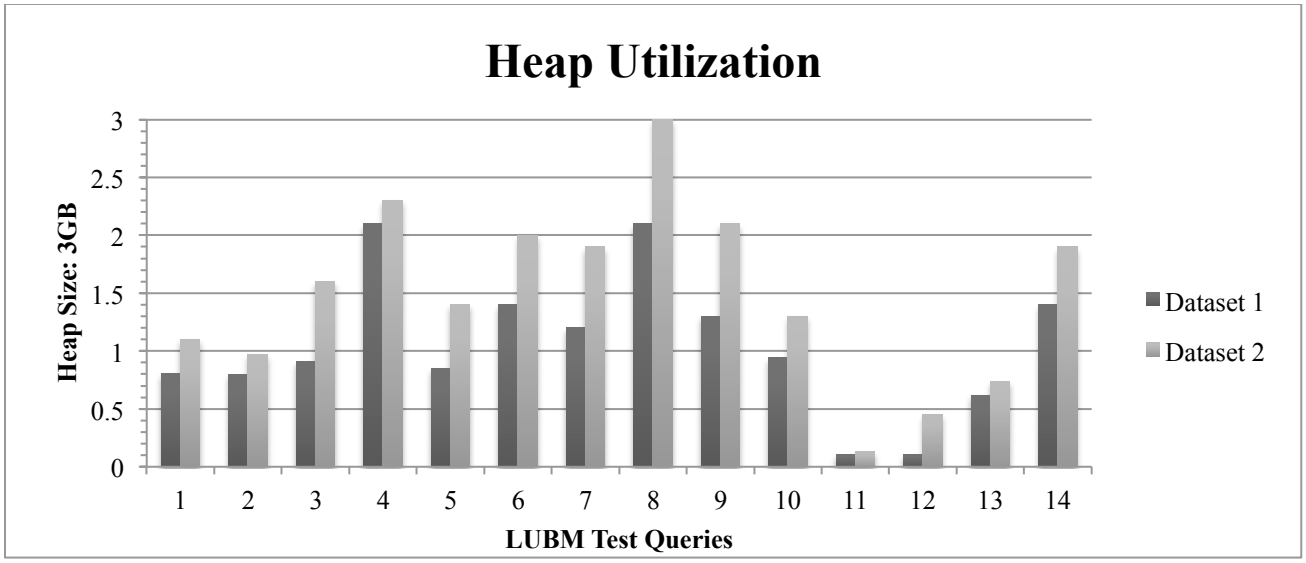

Figure 7: Memory Utilization for 3GB

Above figure shows the Heap utilization of the system to answer query for both the data sets when the heap size of JVM is set to 3GB.

6.2.3 HEAP SIZE: 2GB

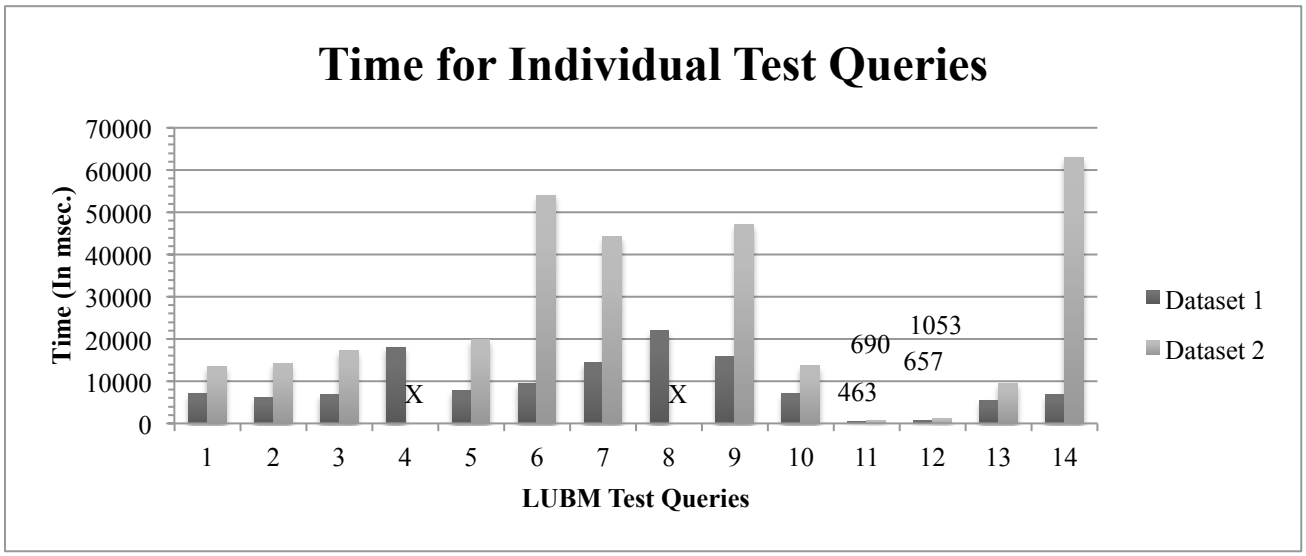

Figure 8: Time for Test Queries v/s 2GB Memory Space

Above figure shows the response time of the system to answer query for both the data sets when the heap size of JVM is set to 2GB.

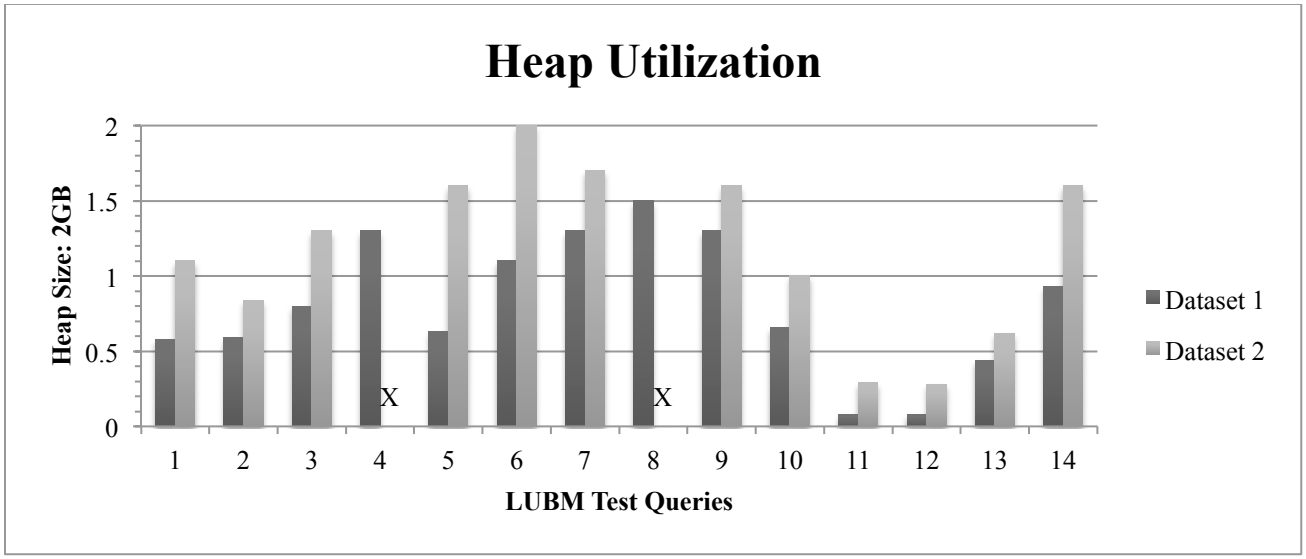

Figure 9: Memory Utilization for 2GB

Above figure shows the Heap utilization of the system to answer query for both the data sets when the heap size of JVM is set to 2GB.

6.2.4 HEAP SIZE: 1GB

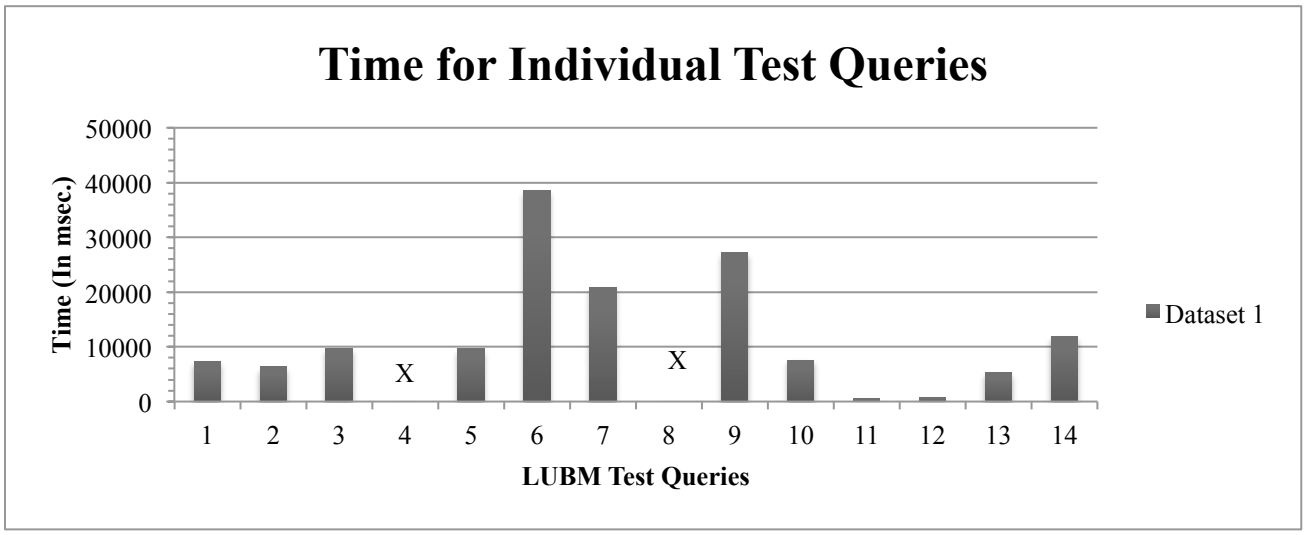

Figure 10: Time for Test Queries v/s 1GB Memory Space

Above figure shows the response time of the system to answer query for both the data sets when the heap size of JVM is set to 1GB.

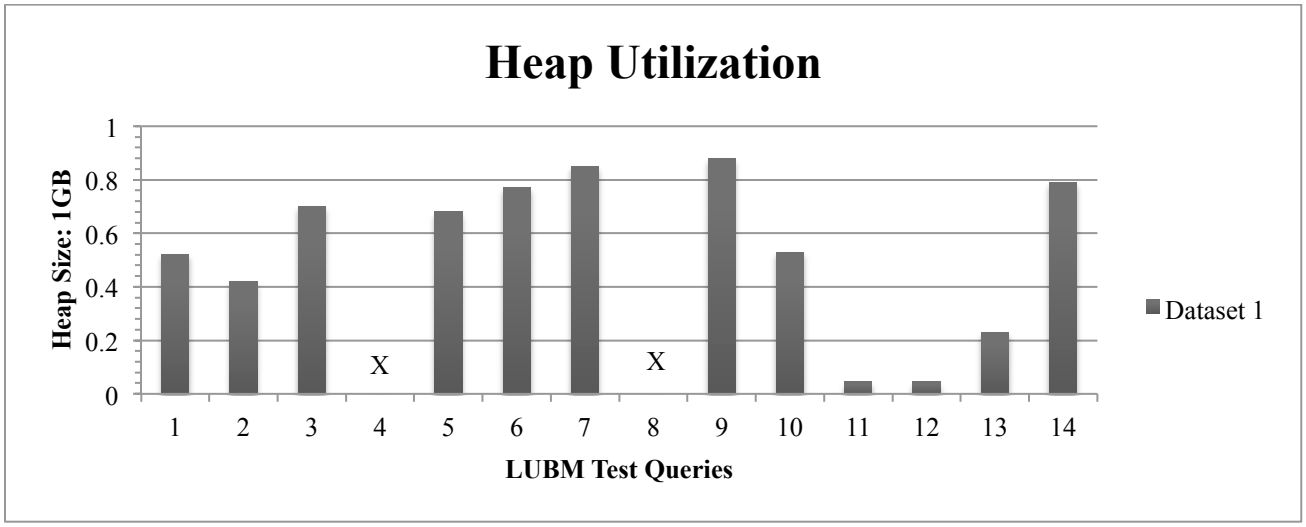

Figure 11: Memory Utilization for 1GB

Above figure shows the Heap utilization of the system to answer query for both the data sets when the heap size of JVM is set to 1GB.

6.3 RESPONSE TIME FOR INDIVIDUAL LUBM TEST QUERIES:

6.3.1 Query 1

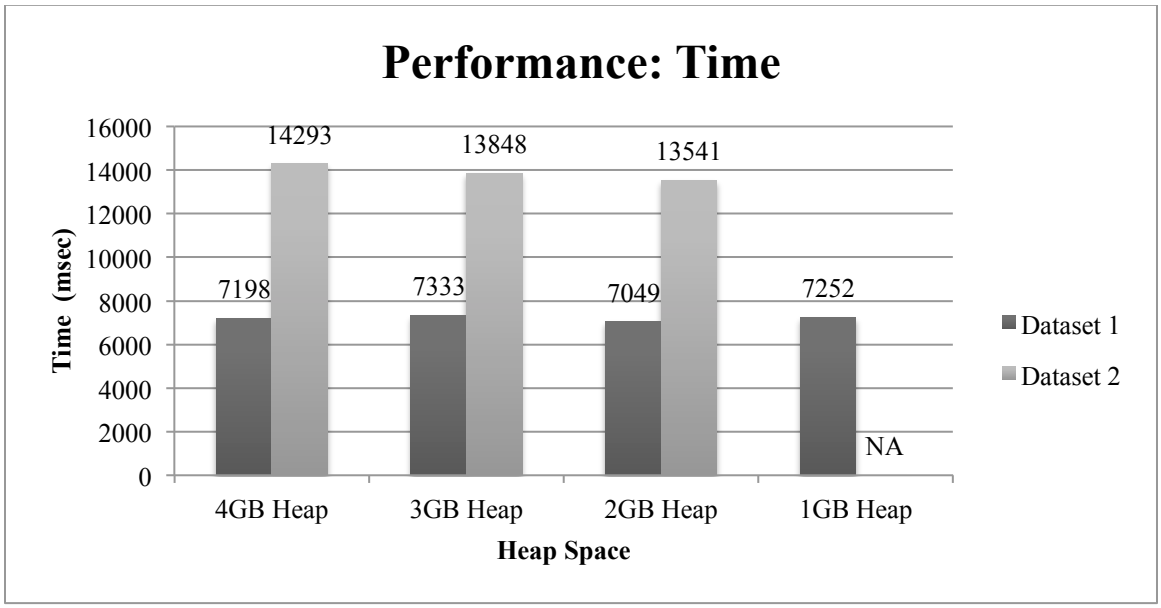

Figure 12: Time v/s Memory for Query 1

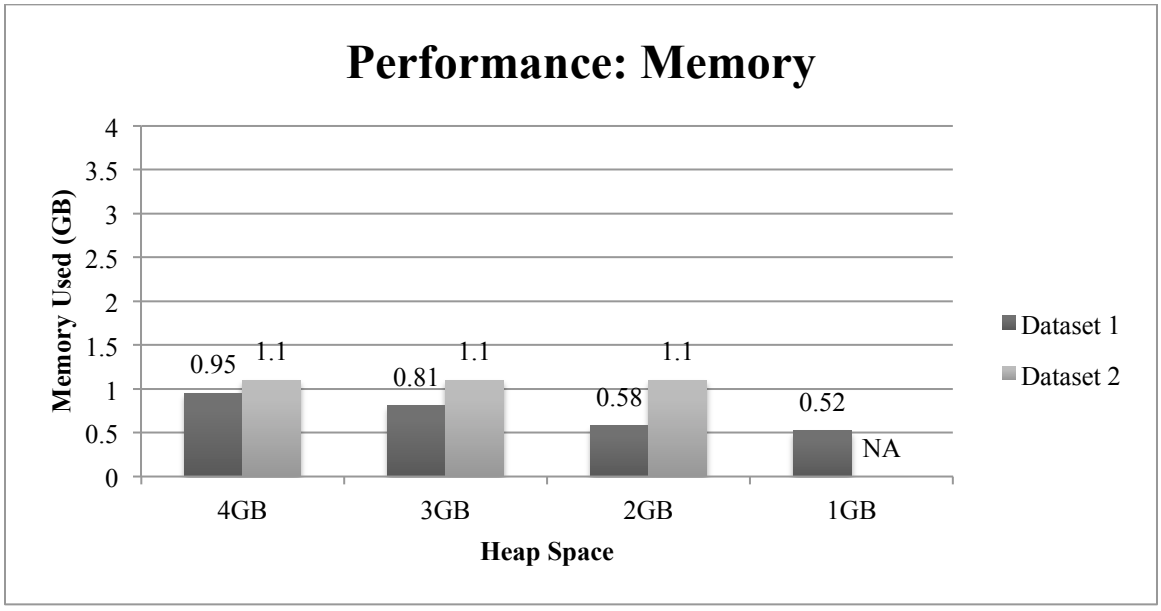

Figure 13: Memory Utilization for Query 1

6.3.2 Query 2

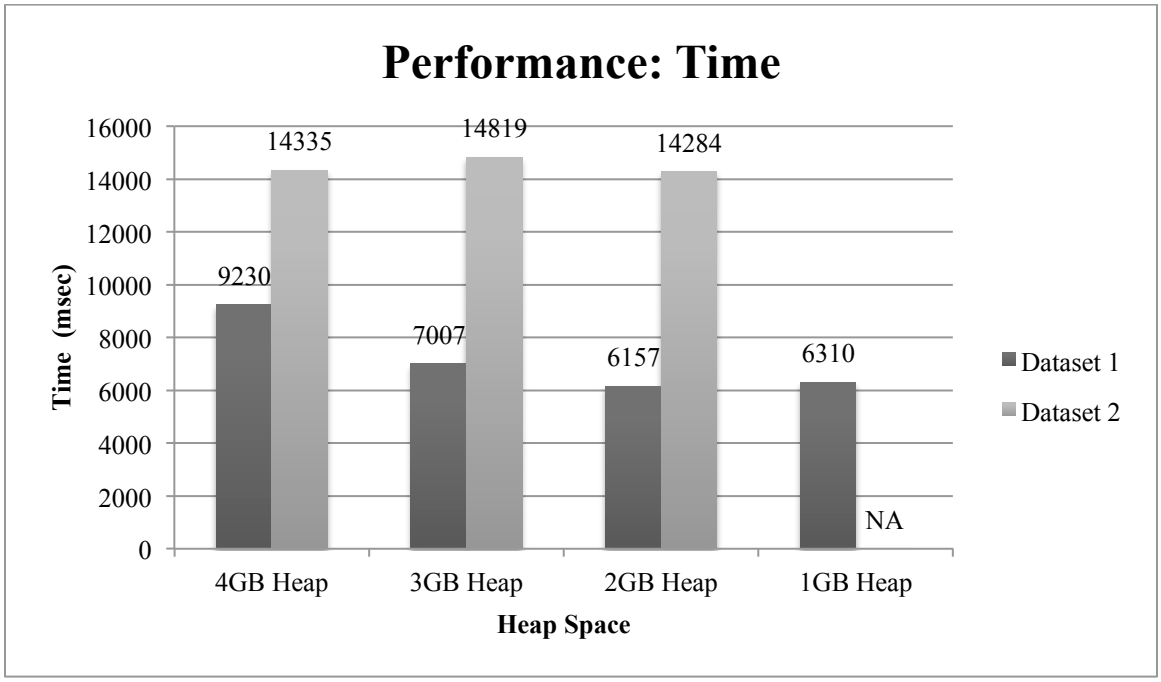

Figure 14: Time v/s Memory for Query 2

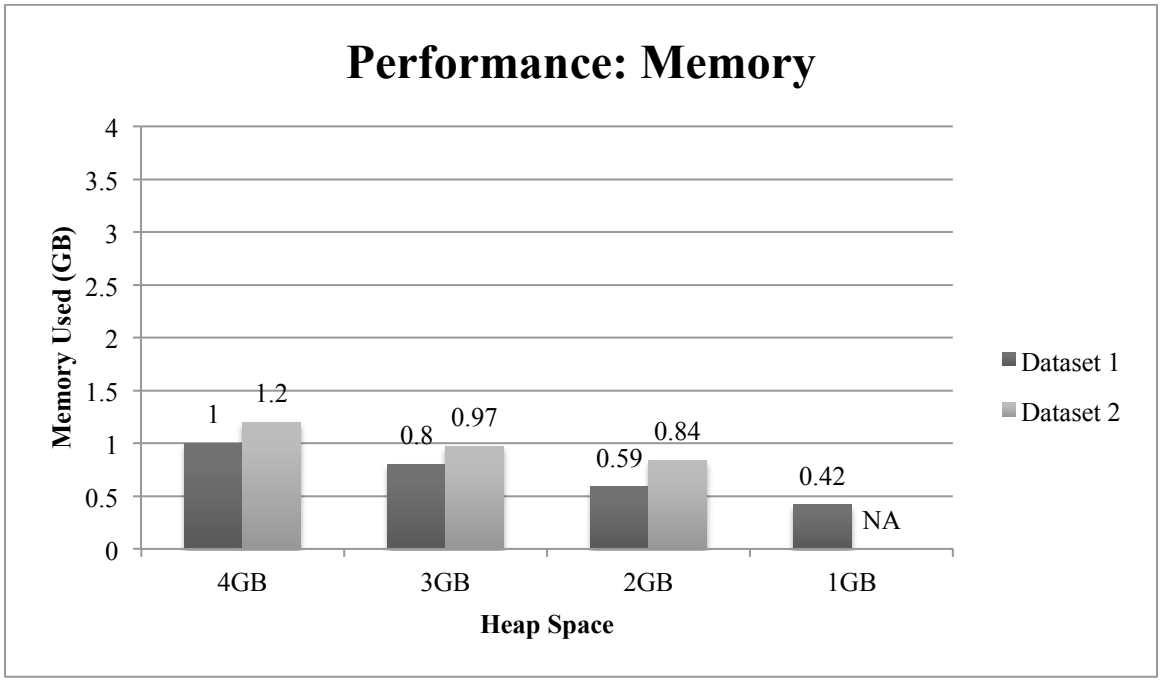

Figure 15: Memory Utilization for Query 2

6.3.3 Query 3

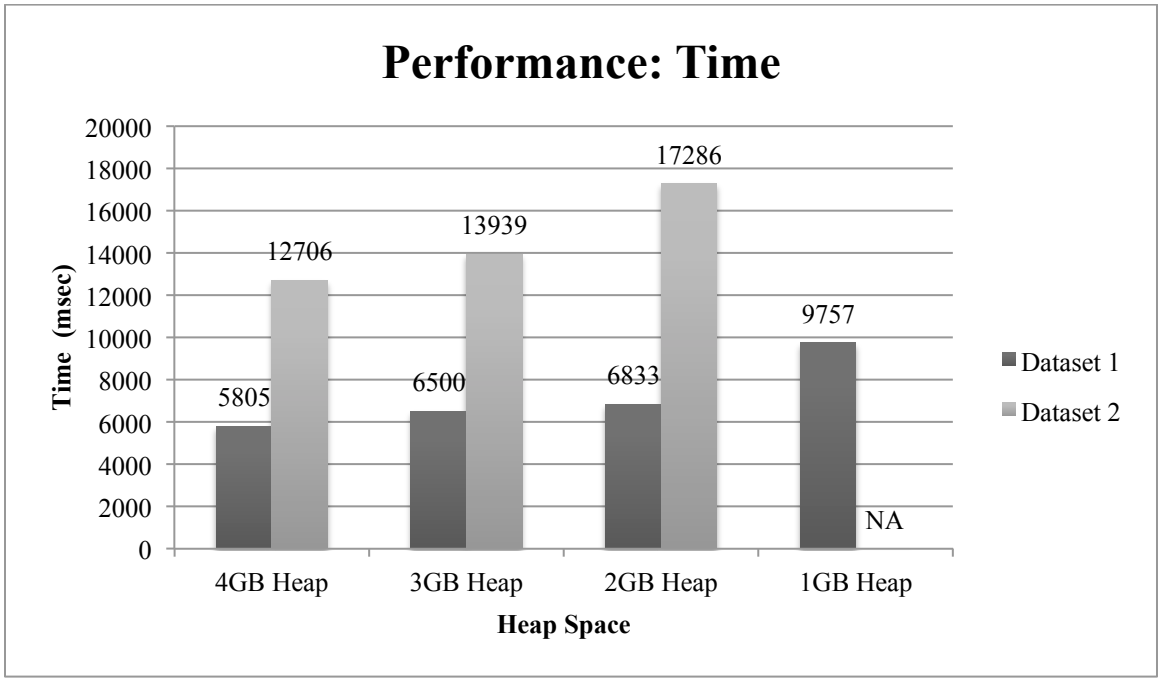

Figure 16: Time v/s Memory for Query 3

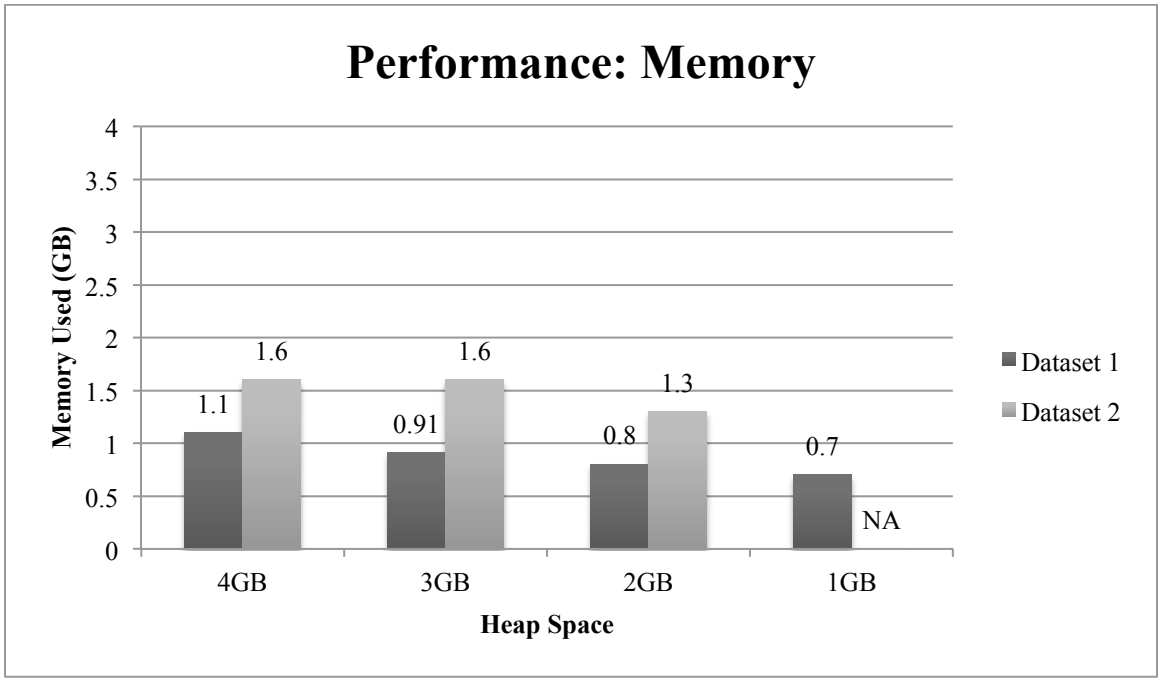

Figure 17: Memory Utilization for Query 3

6.3.4 Query 4

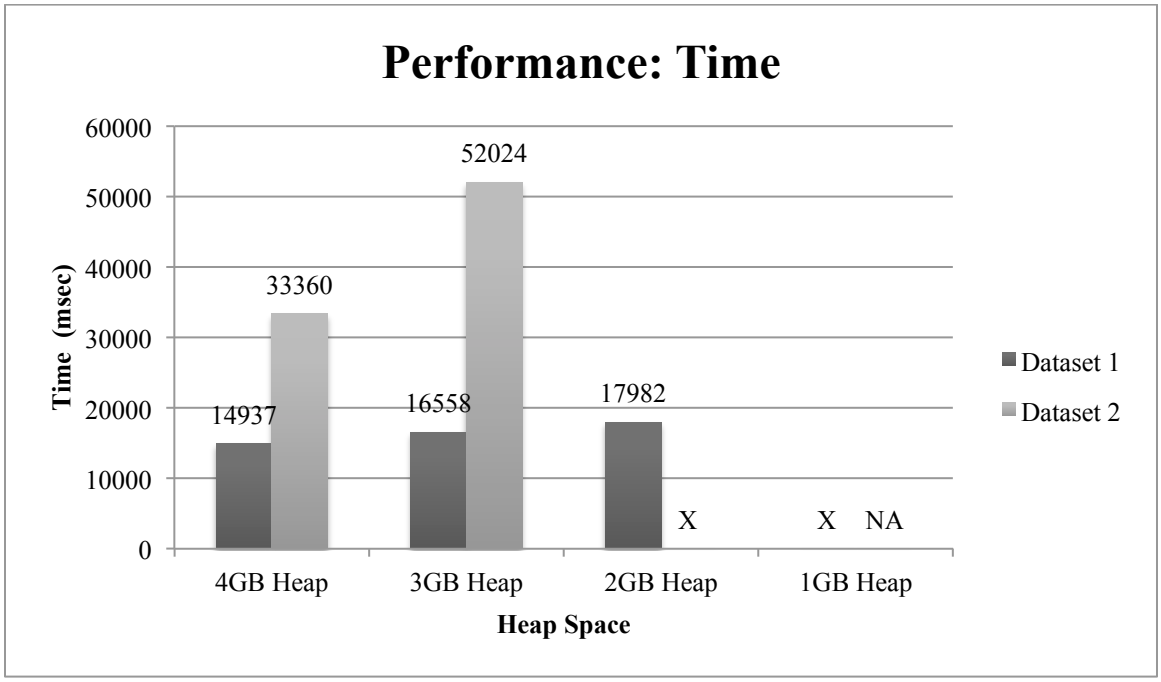

Figure 18: : Time v/s Memory for Query 4

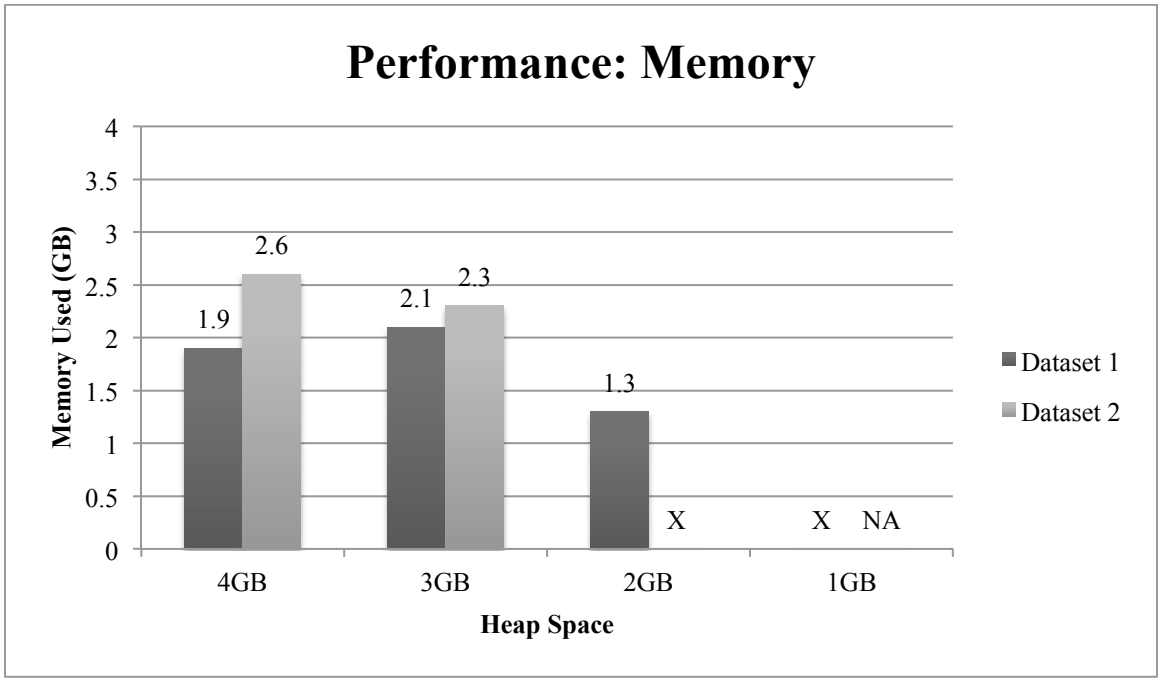

Figure 19: Memory Utilization for Query 4

6.3.5 Query 5

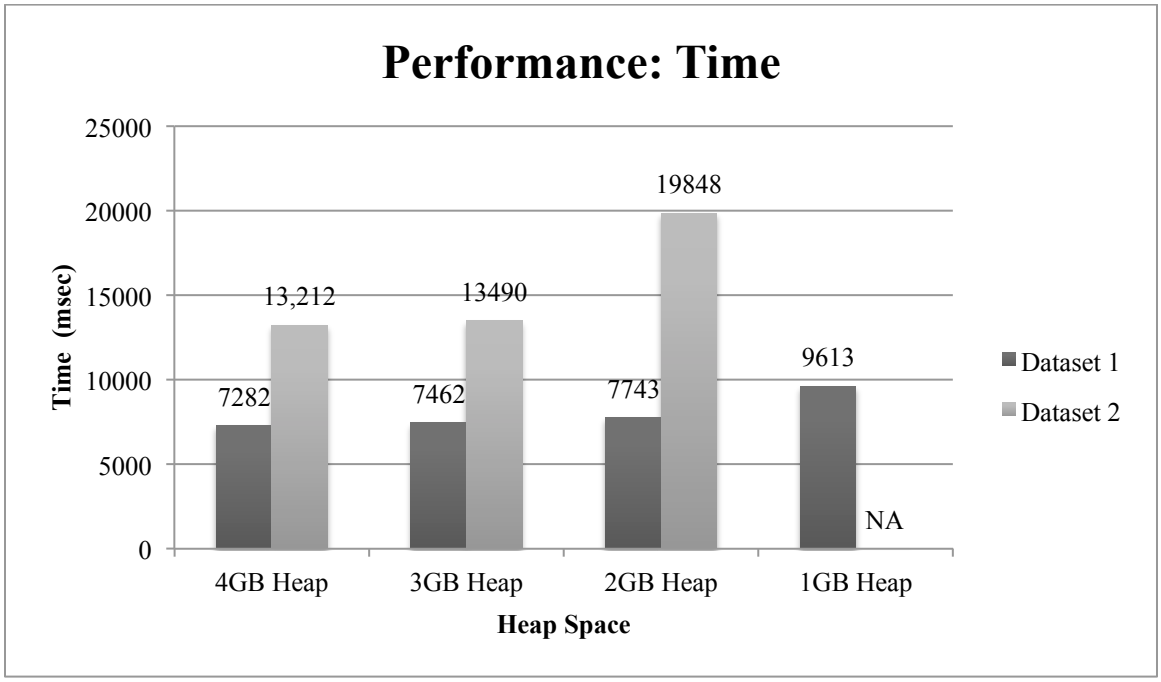

Figure 20: Time v/s Memory for Query 5

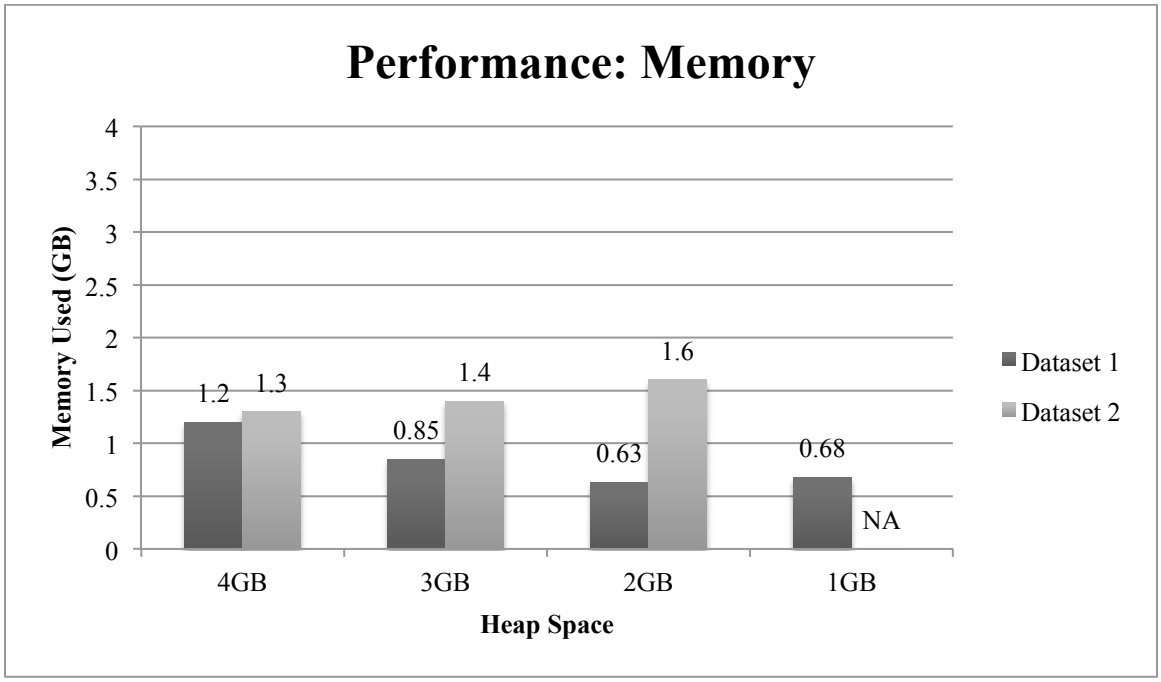

Figure 21: Memory Utilization for Query 5

6.3.6 Query 6

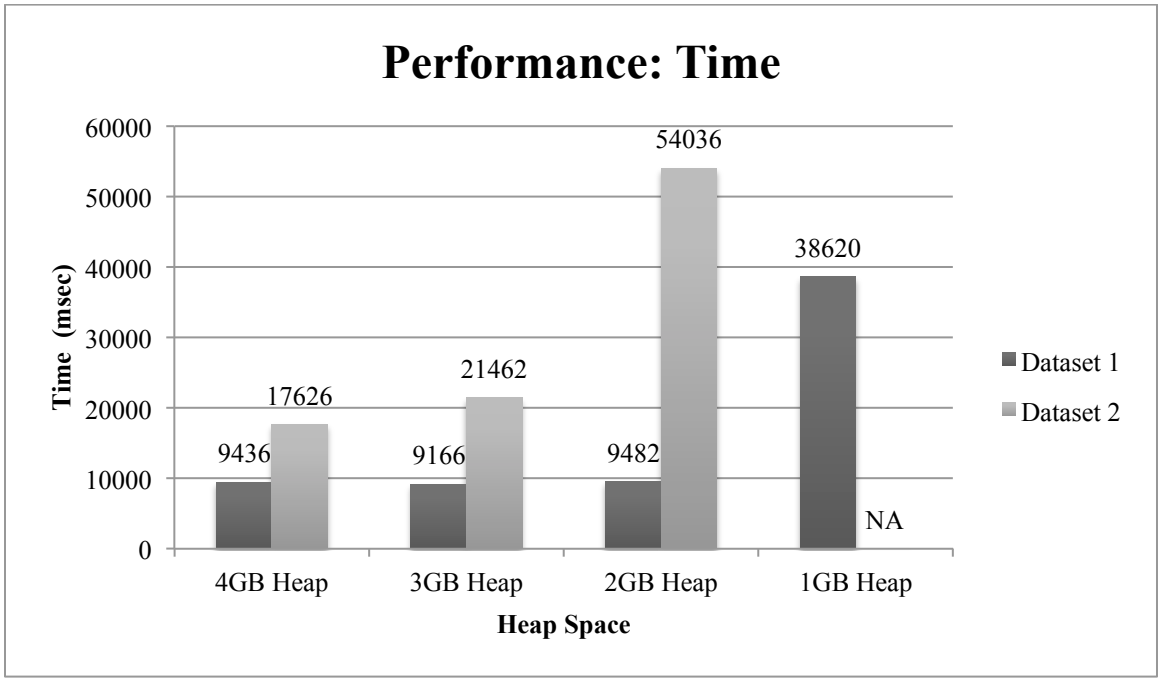

Figure 22: Time v/s Memory for Query 6

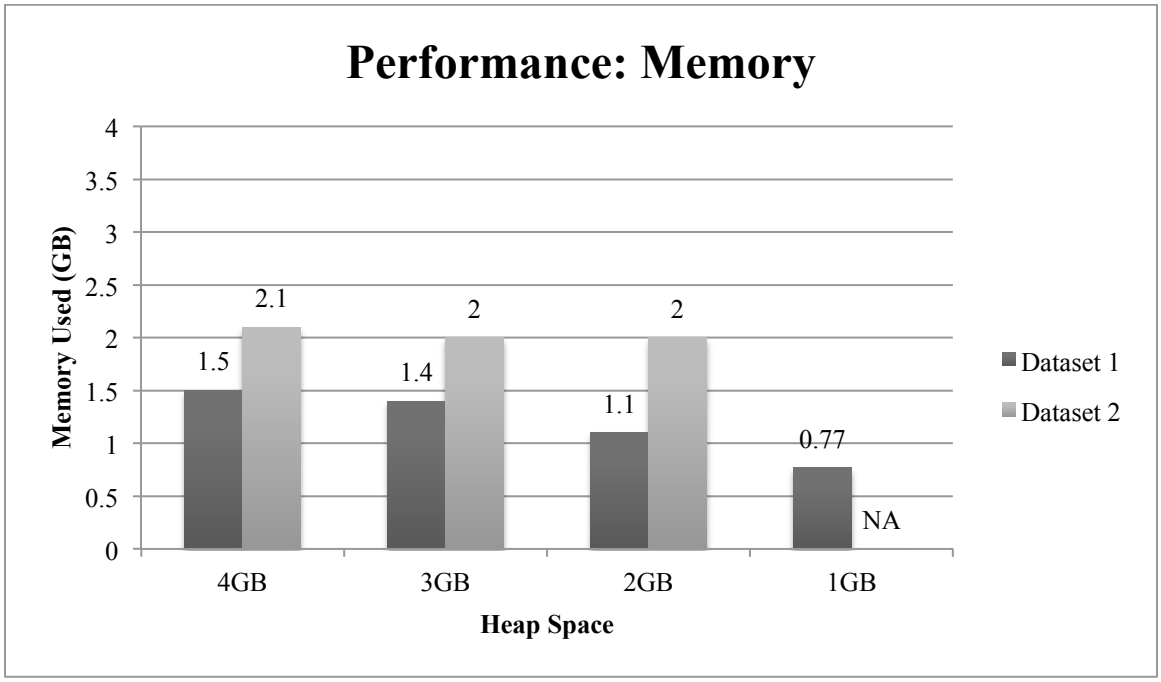

Figure 23: Memory Utilization for Query 6

6.3.7 Query 7

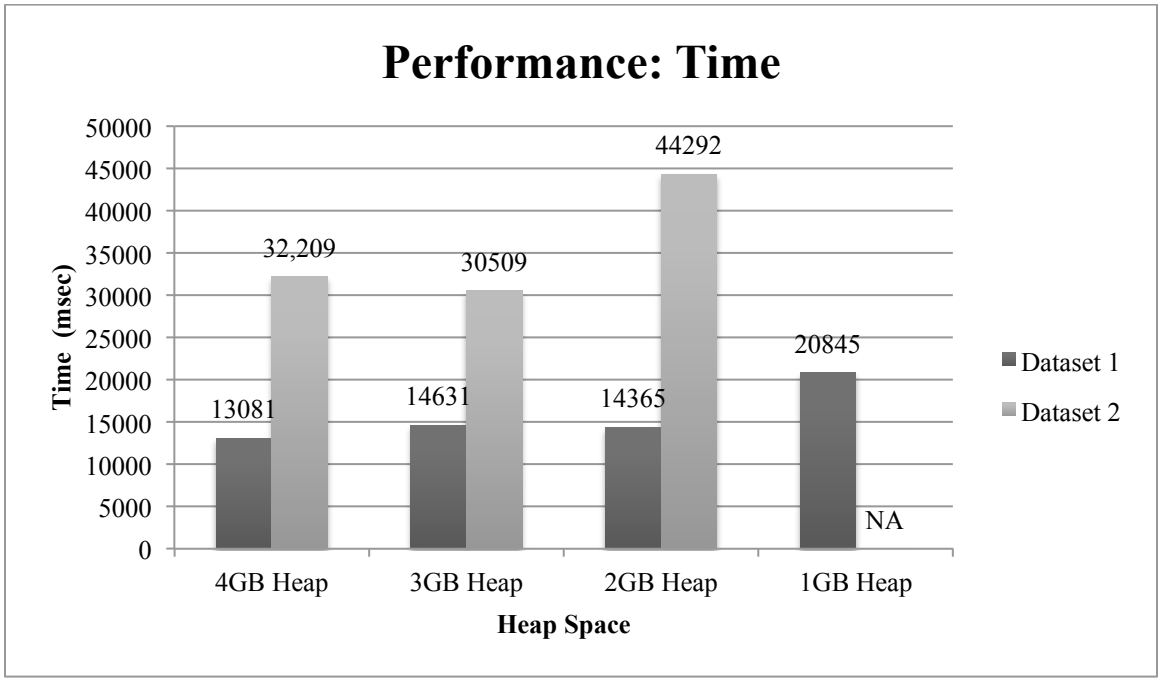

Figure 24: Time v/s Memory for Query 7

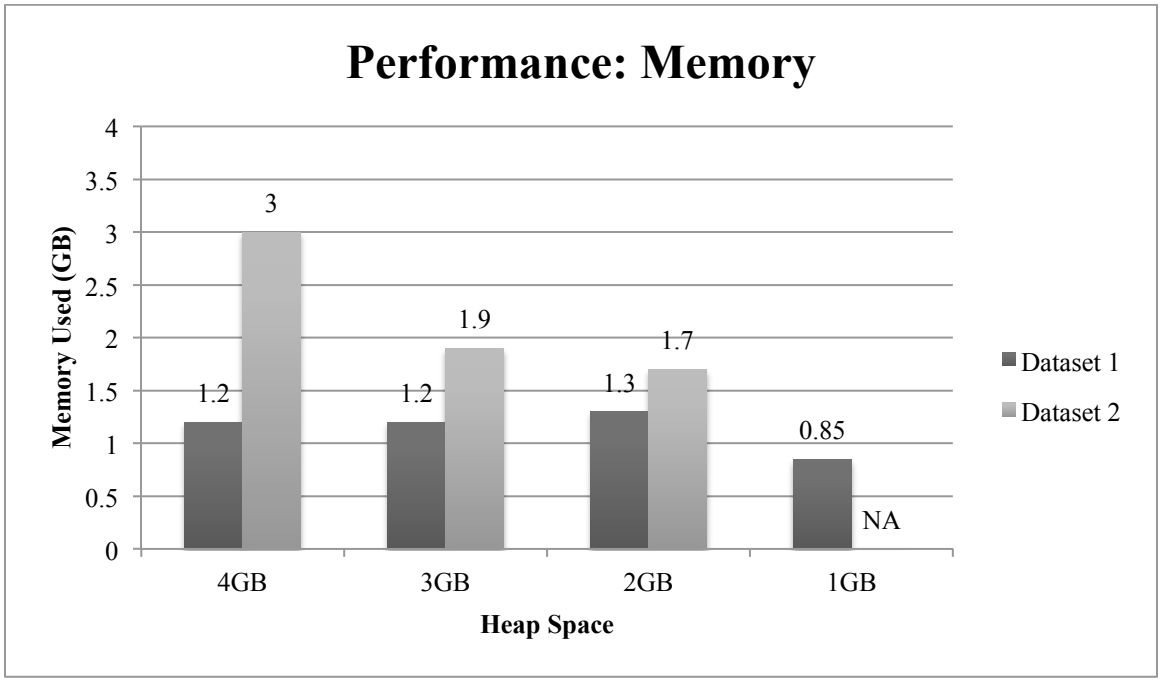

Figure 25: Memory Utilization for Query 7

6.3.8 Query 8

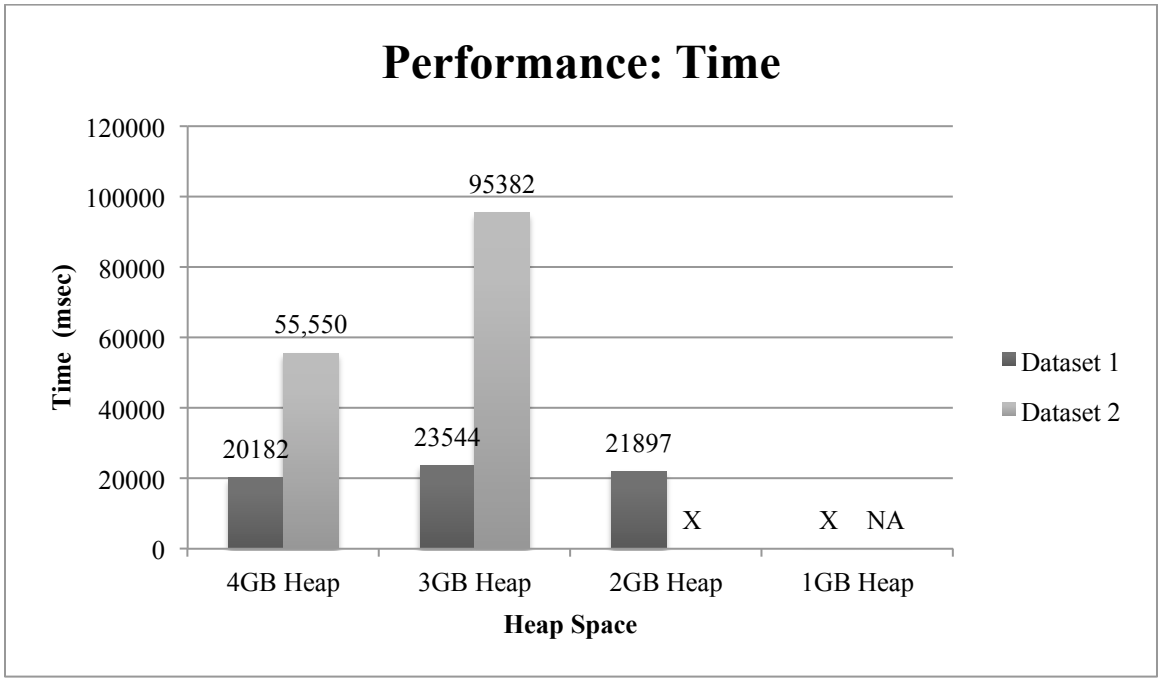

Figure 26: Time v/s Memory for Query 8

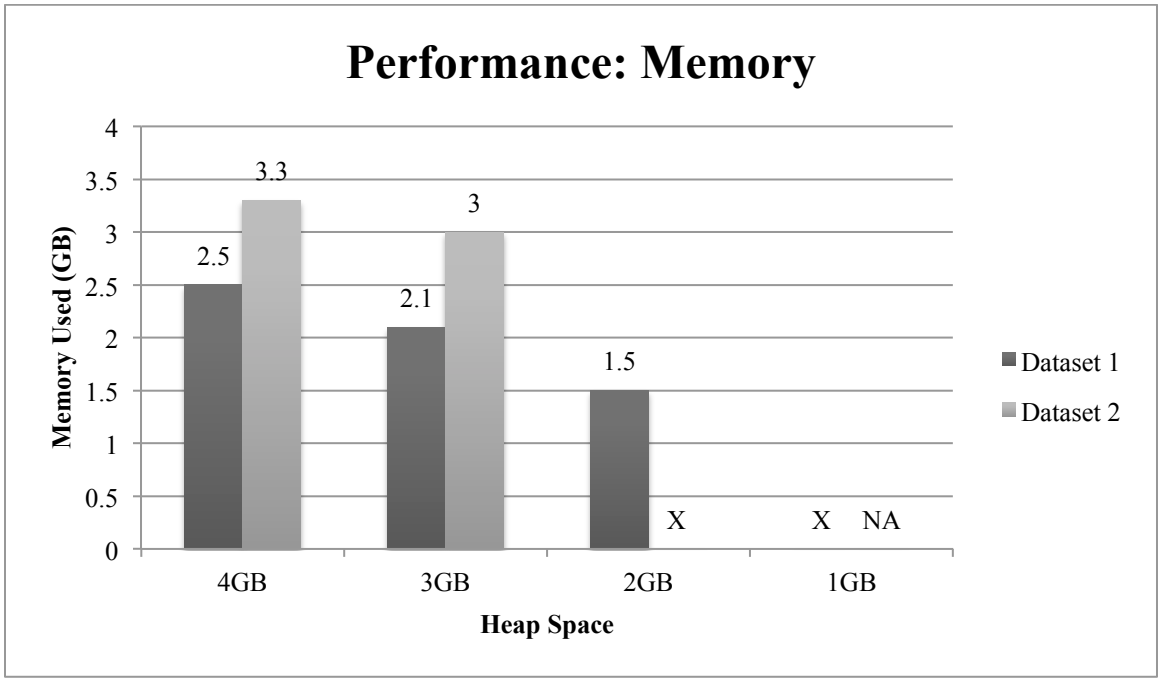

Figure 27: Memory Utilization for Query 8

6.3.9 Query 9

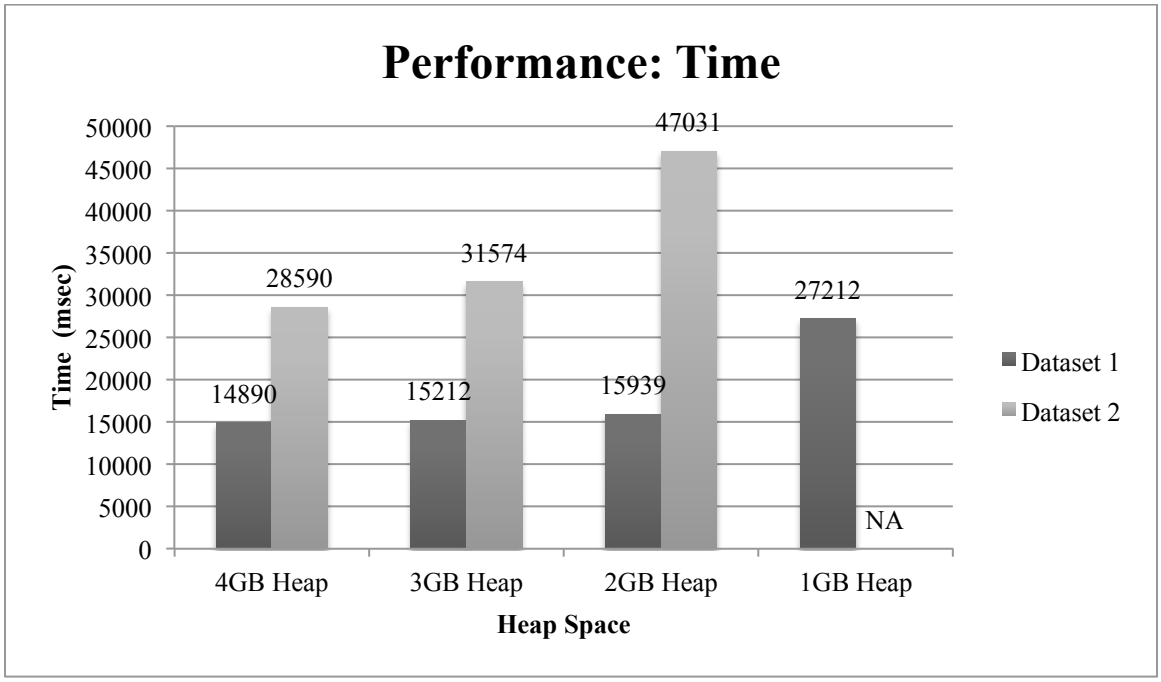

Figure 28: Time v/s Memory for Query 9

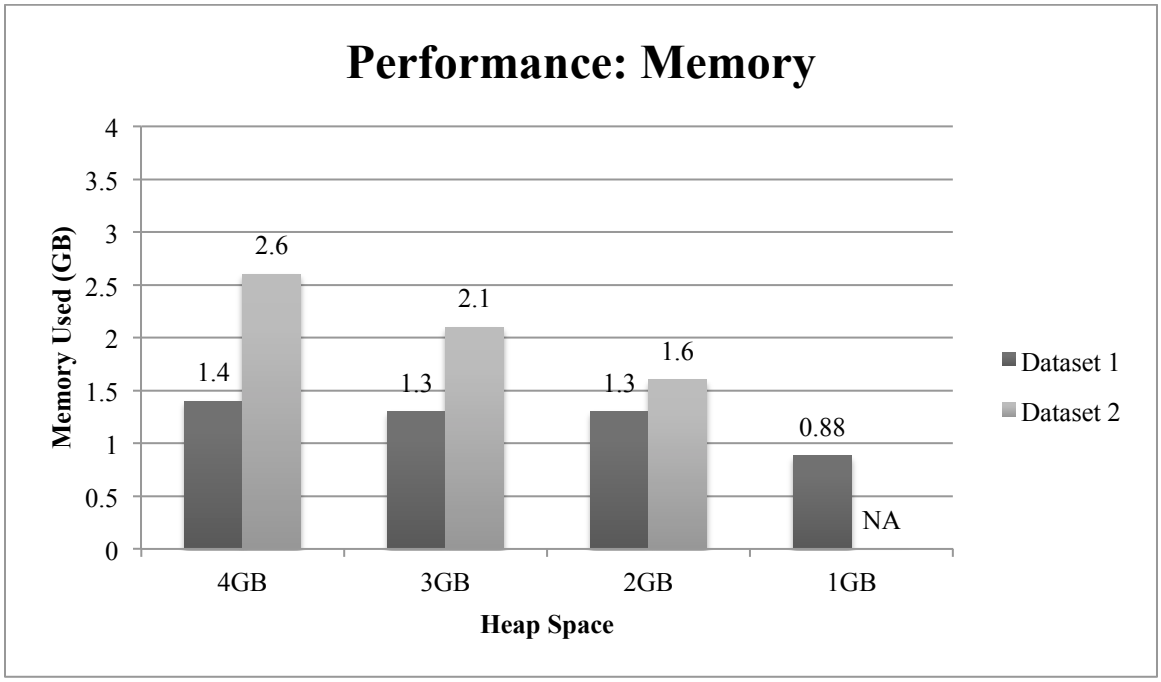

Figure 29: Memory Utilization for Query 9

6.3.10 Query 10

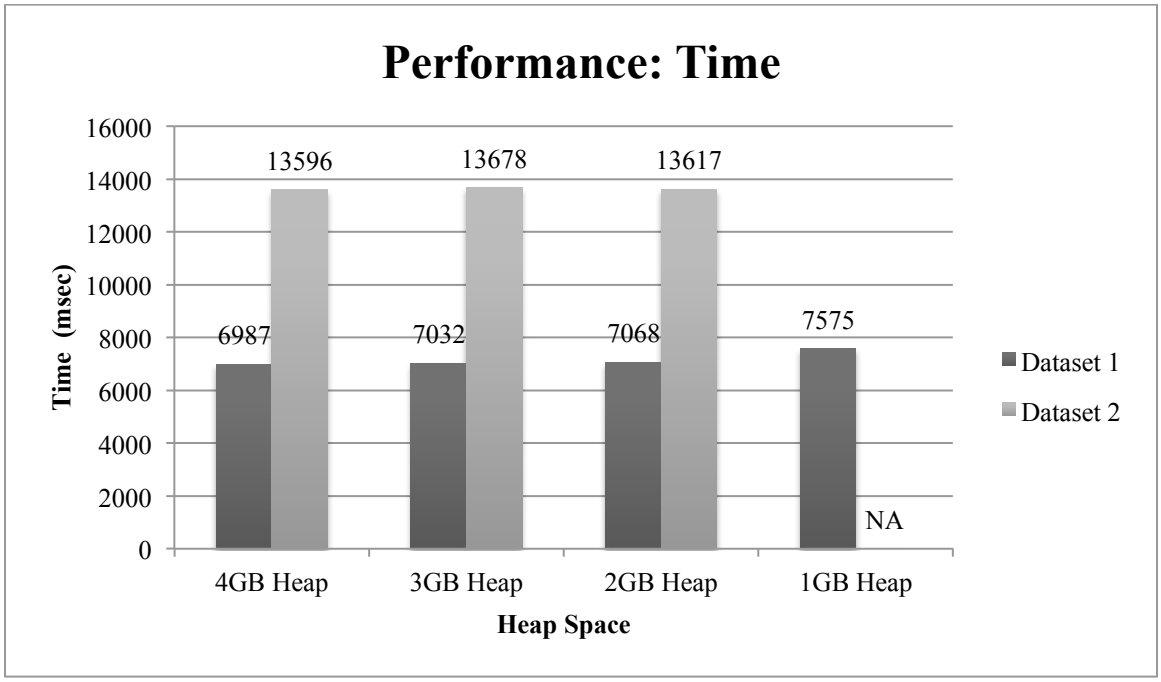

Figure 30: Time v/s Memory for Query 10

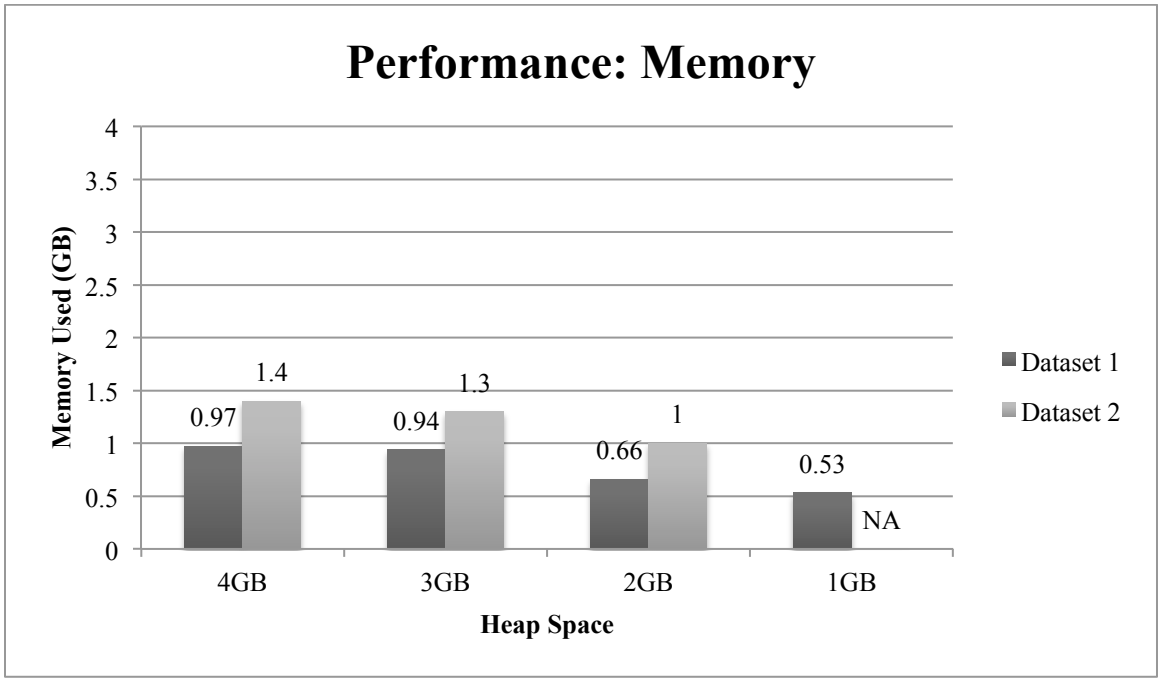

Figure 31: Memory Utilization for Query 10

6.3.11 Query 11

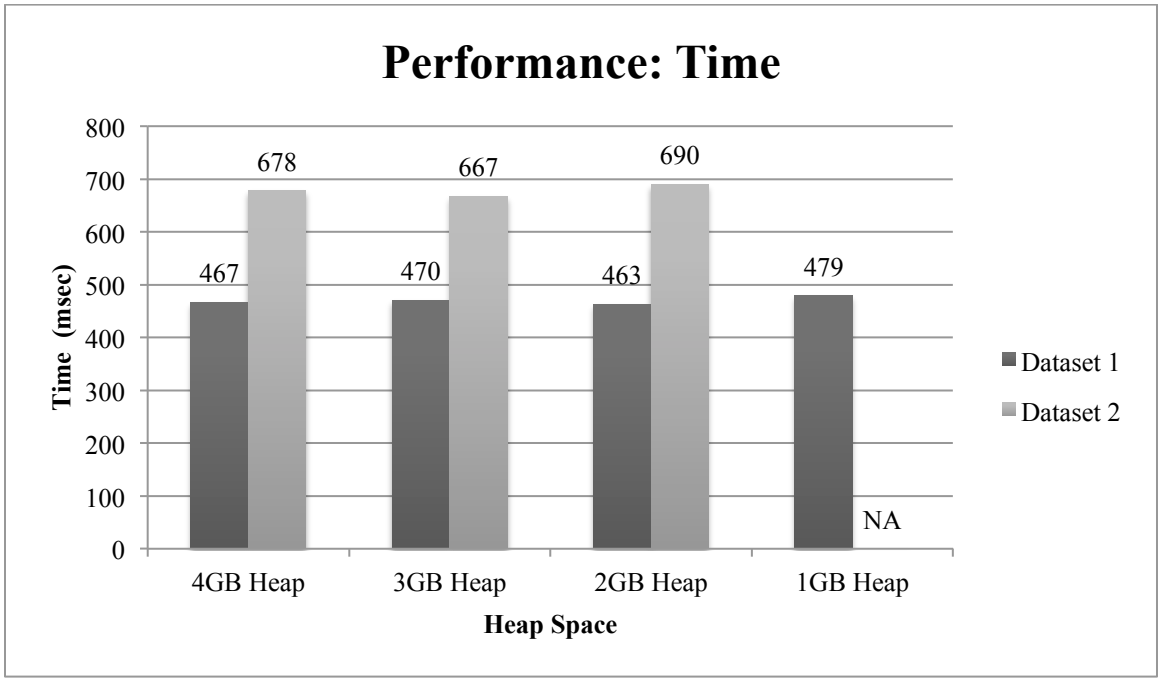

Figure 32: Time v/s Memory for Query 11

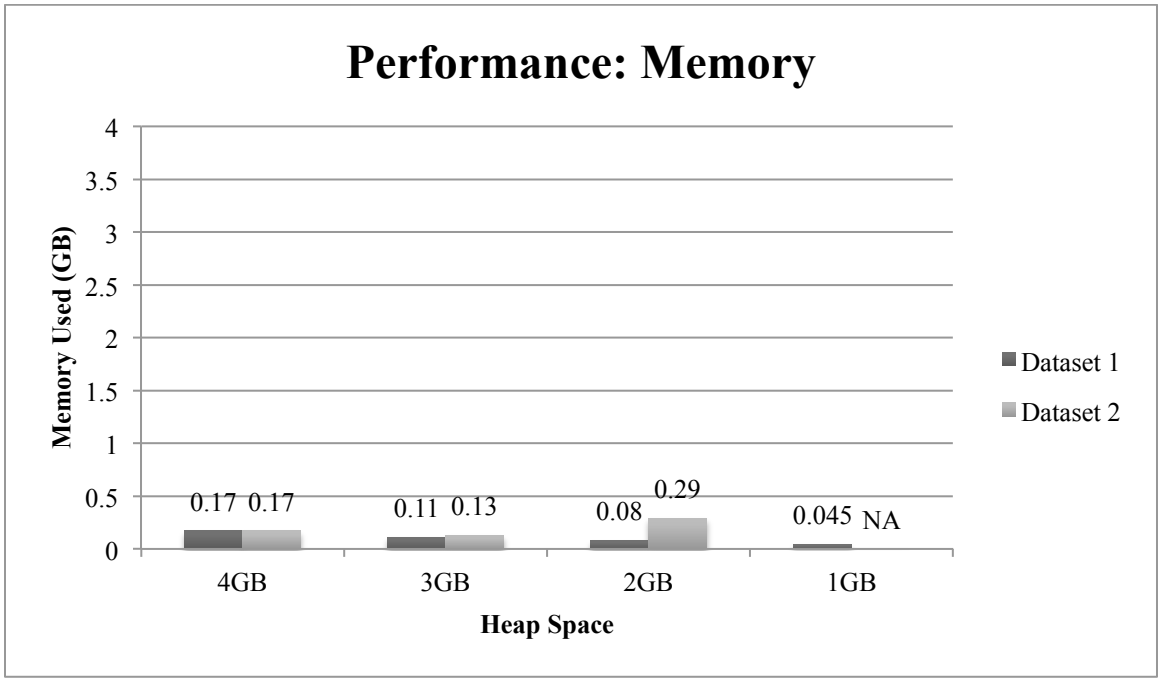

Figure 33: Memory Utilization for Query 11

6.1.12 Query 12

Figure 34: Time v/s Memory for Query 12

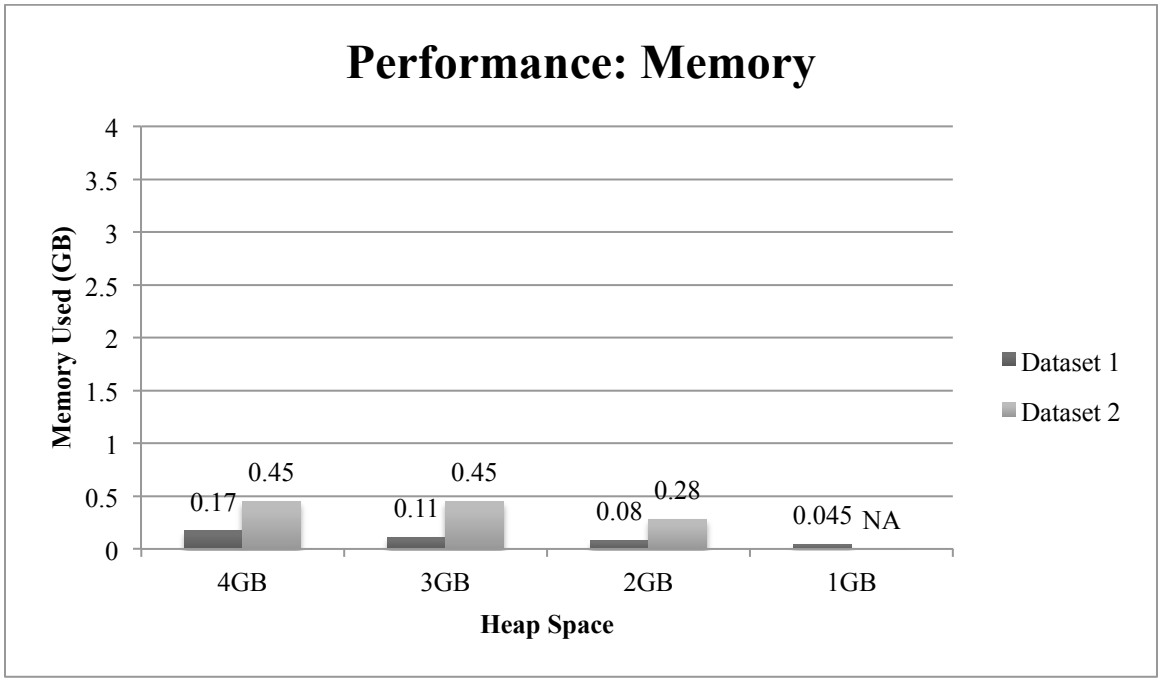

Figure 35: Memory Utilization for Query 12

6.3.13 Query 13

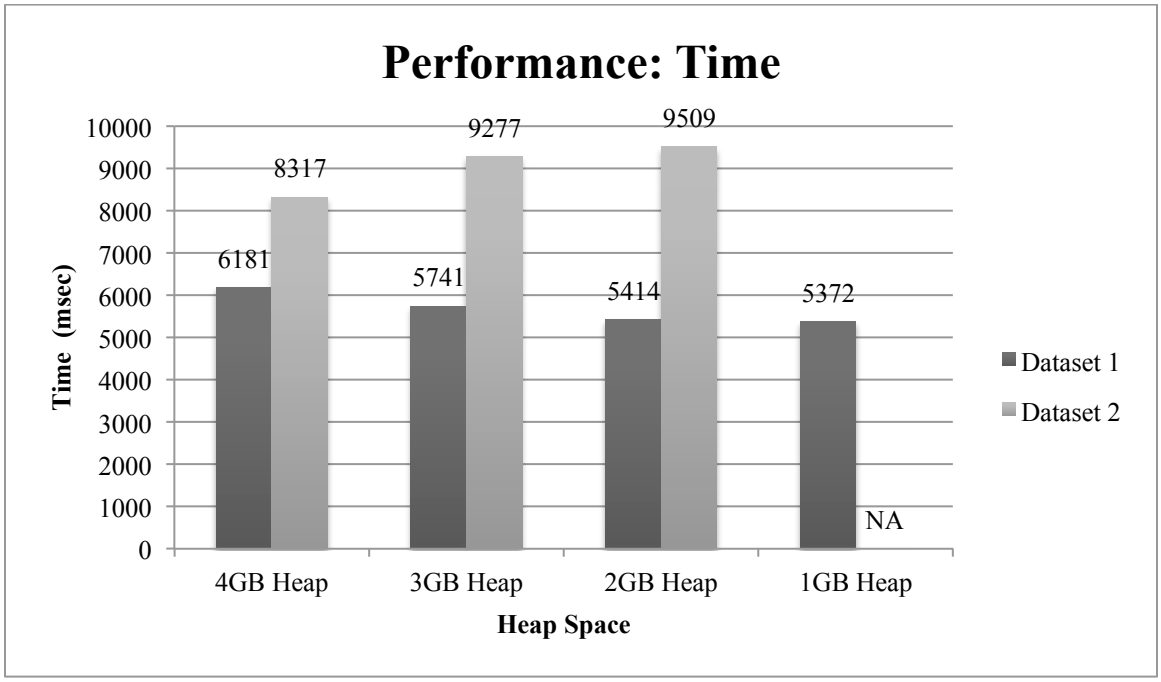

Figure 36: Time v/s Memory for Query 13

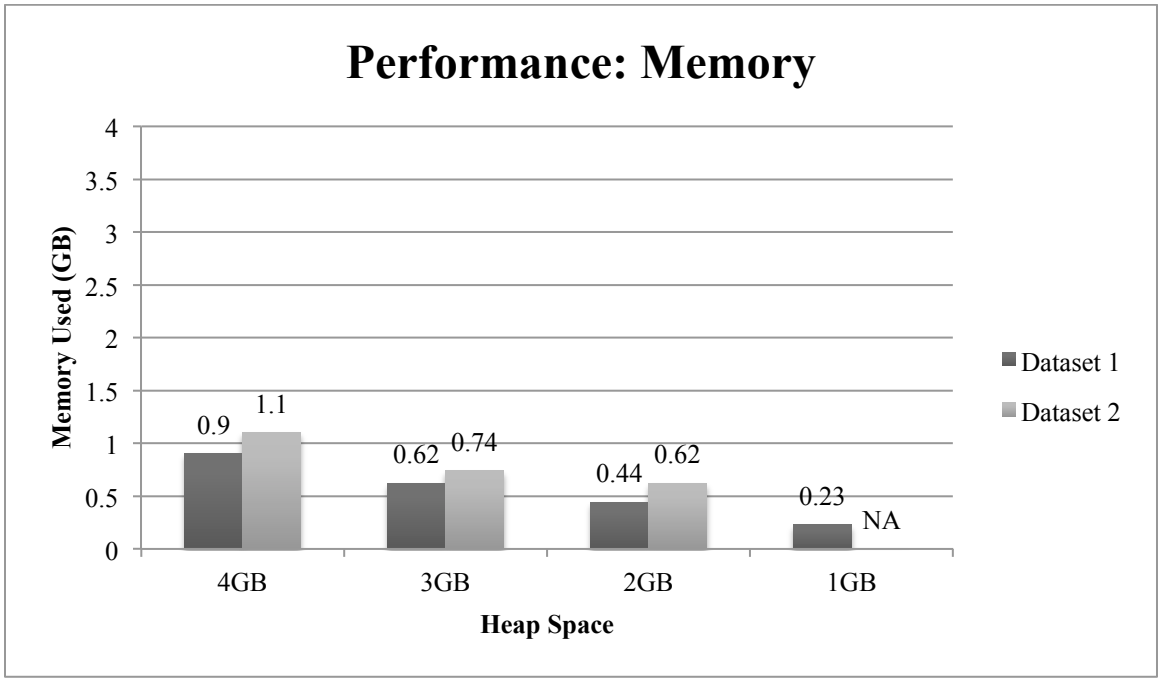

Figure 37: Memory Utilization for Query 13

6.3.14 Query 14

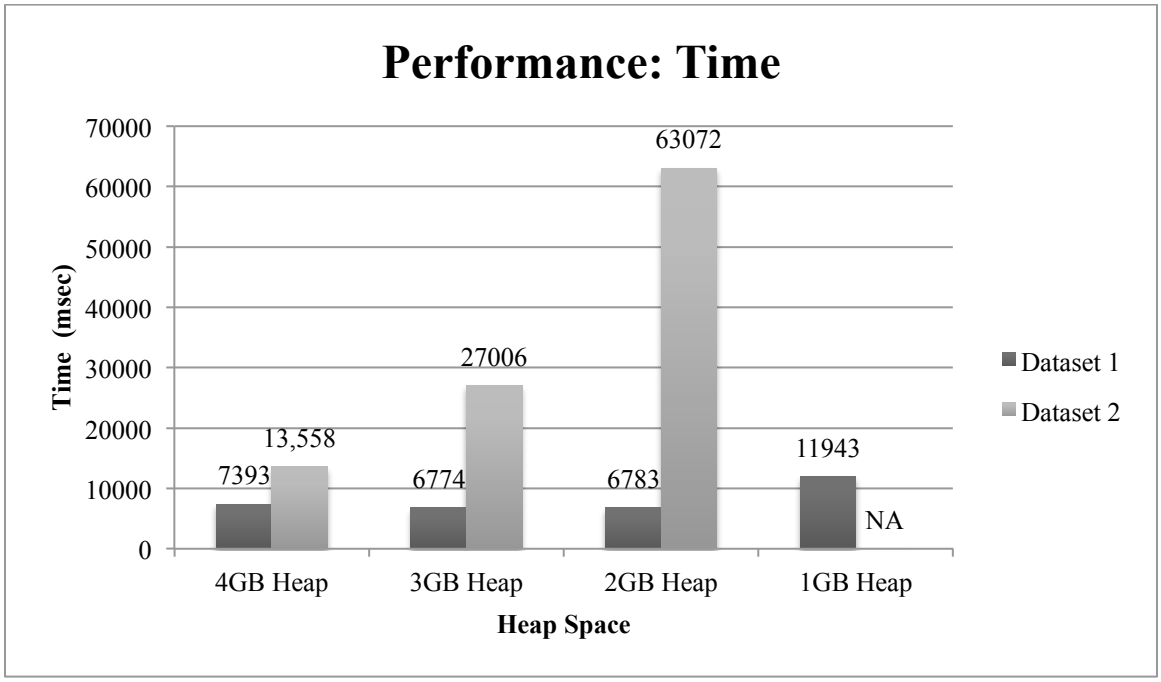

Figure 38: Time v/s Memory for Query 14

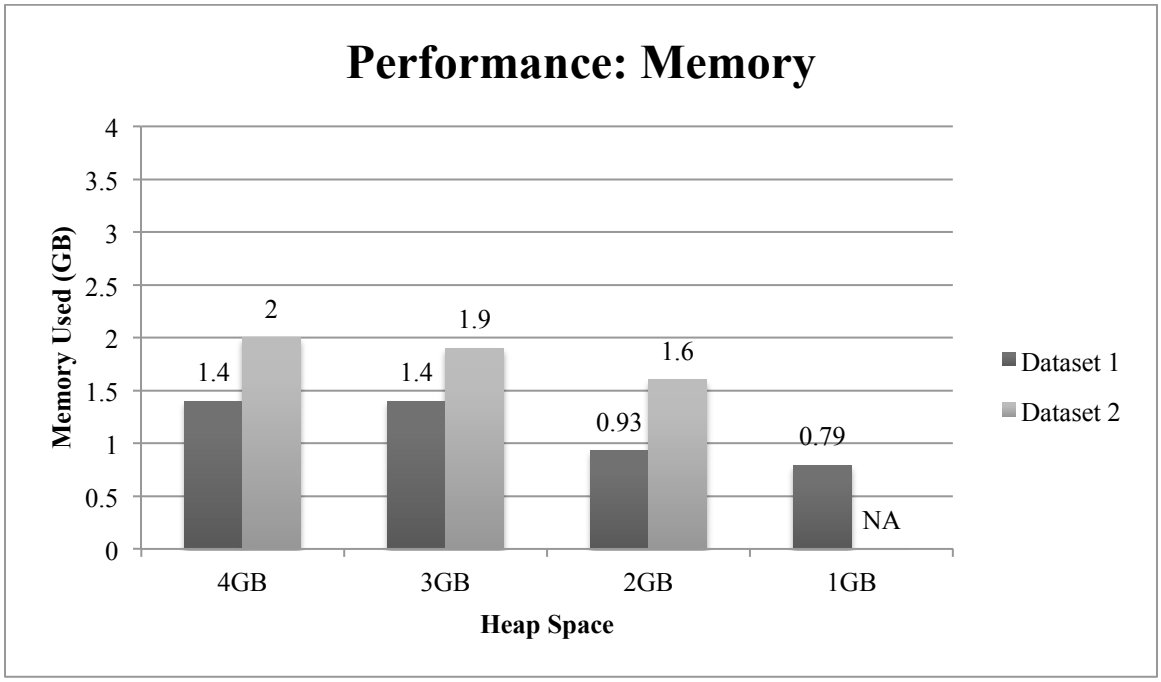

Figure 39: Memory Utilization for Query 14

6.4 QUERY EVALUATION, DATA LOAD AND QUERY EXECUTION TIME OF RGIS:

For RGIS, the total execution time can be split into:

- Query Plan Generation
- Load Files into memory
- Execute Query Plan.

Time required for Query Plan Generation, Loading the required files into memory and execution of Query Plan for all the LUBM Test Queries on our datasets are given below.

6.4.1 Heap Space 4GB:

LUBM Test Query	Data Set 1				Data Set 2			
	Query Plan Generation (In msec.)	Load Files in Memory (In msec.)	Execute Query Plan (In msec.)	Total Execution Time (In msec.)	Query Plan Generation (In msec.)	Load Files in Memory (In msec.)	Execute Query Plan (In msec.)	Total Execution Time (In msec.)
1	7	6627	563	7198	6	1256	1726	14293
2	9	3096	6124	9230	8	5644	8681	14335
3	7	3801	1996	5805	6	714	5559	12706
4	8	5290	9639	14937	6	9420	23933	33360
5	6	2266	5009	7282	6	4040	9165	13212
6	5	11	9419	9436	4	10	17611	17626
7	8	7127	5945	13081	8	14820	17380	32209
8	6	6838	13337	20182	7	14477	41065	55550
9	10	8139	6740	14890	8	15712	12869	28590
10	6	6427	554	6987	5	12241	1349	13596
11	8	315	143	467	11	399	267	678
12	8	555	109	672	7	869	186	1062
13	12	2651	3517	6181	14	4220	4082	8317
14	5	14	7373	7393	4	13	13540	13558

Table 10: Total Execution Time of RGIS for 4GB Heap Space

6.4.2 Heap Space 3GB:

LUBM Test Query	Data Set 1				Data Set 2			
	Query Plan Generation (In msec.)	Load Files in Memory (In msec.)	Execute Query Plan (In msec.)	Total Execution Time (In msec.)	Query Plan Generation (In msec.)	Load Files in Memory (In msec.)	Execute Query Plan (In msec.)	Total Execution Time (In msec.)
1	7	6748	577	7333	6	12068	1772	13848
2	10	3045	3952	7007	7	5424	9386	14819
3	29	3901	2569	6500	7	7373	6557	13939
4	41	5530	10985	16558	6	15810	36206	52024
5	6	2210	5244	7462	6	3911	9571	13490
6	6	13	9146	9166	6	11	21444	21462
7	8	7575	7047	14631	6	13559	16943	30509
8	7	6277	17260	23544	6	15005	80369	95382
9	8	8406	6797	15212	10	15365	16198	31574
10	7	6471	553	7032	6	11865	1805	13678
11	7	316	146	470	8	400	257	667
12	8	553	106	667	7	845	187	1040
13	11	2633	3095	5741	19	4721	4536	9277
14	4	11	6758	6774	11	36	26958	27006

Table 11: Total Execution Time of RGIS for 3GB Heap Space

6.4.3 Heap Space 2GB:

LUBM Test Query	Data Set 1				Data Set 2			
	Query Plan Generation (In msec.)	Load Files in Memory (In msec.)	Execute Query Plan (In msec.)	Total Execution Time (In msec.)	Query Plan Generation (In msec.)	Load Files in Memory (In msec.)	Execute Query Plan (In msec.)	Total Execution Time (In msec.)
1	6	6356	685	7049	7	11846	1687	13541
2	9	2960	3187	6157	10	5502	8770	14284
3	6	3290	2906	6833	5	8728	8551	17286
4	47	5167	12767	17982	NA*	NA*	NA*	NA*
5	6	2169	5567	7743	5	3882	15960	19848
6	5	11	9464	9482	36	103	53896	54036
7	8	7716	6640	14365	7	14098	30185	44292
8	9	7755	14133	21897	NA*	NA*	NA*	NA*
9	42	8323	7574	15939	10	15628	31392	47031
10	6	6401	659	7068	6	11922	1689	13617
11	8	309	146	463	9	402	278	690
12	6	543	106	657	8	860	185	1053
13	11	2320	3082	5414	13	4383	5112	9509
14	4	10	6768	6783	48	31	62992	63072

Table 12: Total Execution Time of RGIS for 2GB Heap Space

*For the given Heap Space, RGIS was not able to evaluate the LUBM Test Query.

6.5 COMPARISON WITH JENA TDB:

Jena TDB [20][21] is component of Jena to store and query RDF data. It works with Jena ARQ to provide support for SPARQL. Jena TDB is a high performance RDF store on a single machine. It uses custom B+ trees and memory mapped IO. Thus, it can support up to 1.7[22] Billion Triples.

To get better understanding of the performance of RGIS, we compare it with Jena TDB. The comparison will be with respect to:

- Space required to store the RDF/OWL data.
- Response time for both the systems to evaluate the 14 LUBM Test Queries.

Below are the details of our findings:

6.5.1 Space comparison with JENA TDB:

Dataset	Actual Size	RGIS	Jena TDB
Dataset 1	3.42 GB	1.15 GB	6.97 GB
Dataset 2	6.90 GB	2.33 GB	13.34

Table 13: Space Comparison – RGIS v/s Jena TDB

As seen from the table above, RGIS requires 66% less storage space whereas Jena TDB requires an additional 100% storage space of that of original data.

Hence, when compared directly with RGIS, Jena TDB requires 600% more storage space than RGIS to store the same data.

6.5.2 Response time to evaluate LUBM Test Queries:

6.5.2.1 Response Time for Dataset 1:

For a fair comparison, we used 2GB heap space for both the systems when comparing the response time for Dataset 1. Below is the comparison graph for the same.

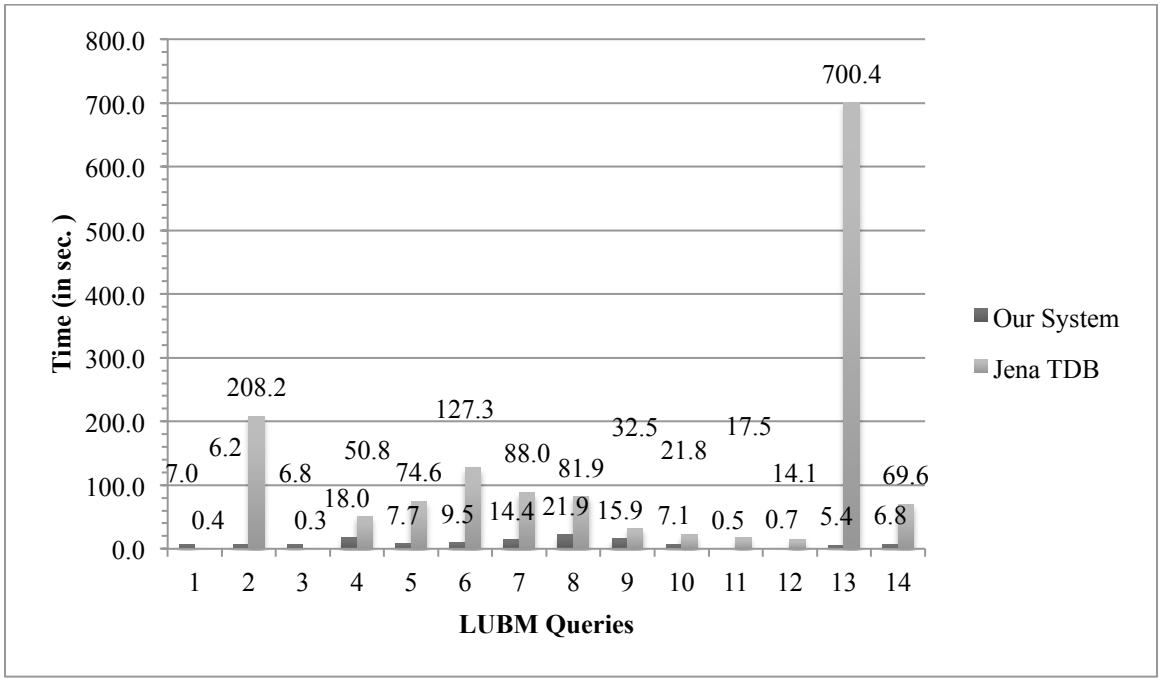

Figure 40: Performance – RGIS v/s Jena TDB for Data Set 1 (41 Million Triples)

6.5.2.2 Response Time for Dataset 2:

For a fair comparison, we used 3GB heap space for both the systems when comparing the response time for Dataset 2. Below is the comparison graph for the same.

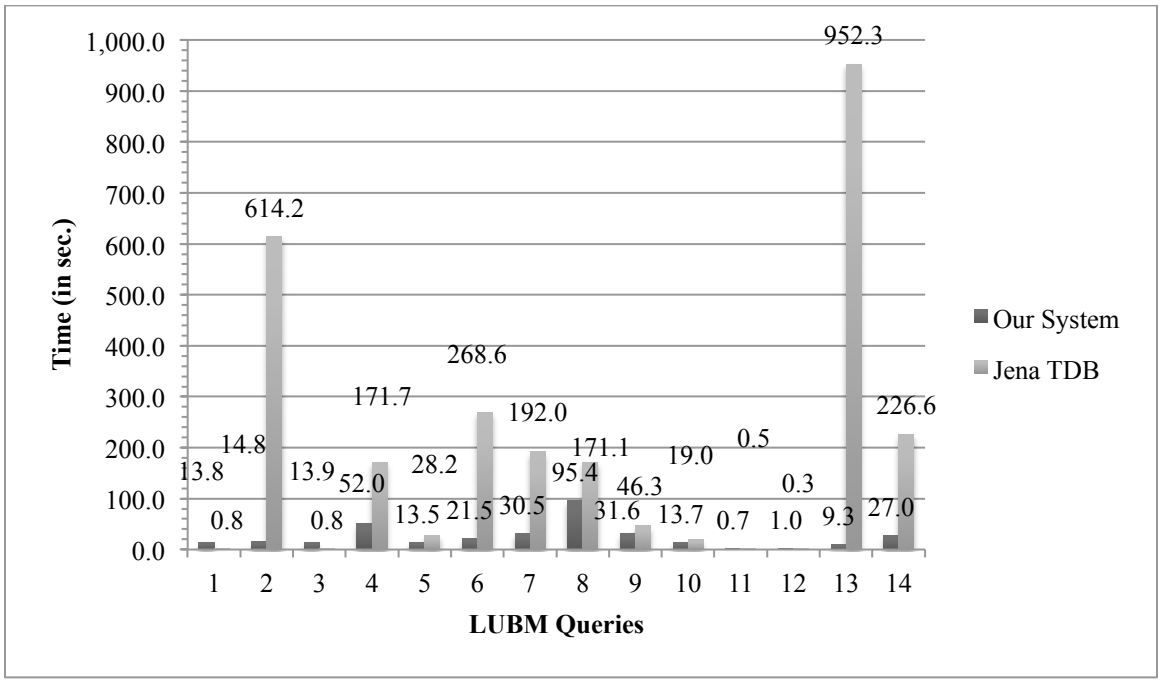

Figure 41: Performance – RGIS v/s Jena TDB for Data Set 2 (83 Million Triples)

6.5.2.3 Total query time

Now, we compare the total time required to execute the LUBM test queries for both the datasets on RGIS and Jena TDB. Below is the table that summarizes the same.

Dataset	RGIS	Jena TDB
	(Time in sec.)	(Time in sec.)
Dataset 1	127.832	1487.283
Dataset 2	338.715	2692.436

Table 14: Total Execution Time

6.6 COMPARISON WITH OWLIM-LITE:

OWLIM Lite is the in-memory model of OWLIM. Previous research [1][4] has shown that OWLIM-Lite cannot scale for more than 10M triples. Our test Data Sets comprised of 41.27M and 82.92M Triples. Hence, as seen in previous research, we were not able to load the datasets into the memory and evaluate the LUBM Test Queries. The Java Heap Space used was 2GB and 3GB respectively.

6.7 COMPARISON WITH OWLIM-SE (BigOWLIM):

We compared performance of RGIS with OWLIM-SE, formerly called as BigOWLIM. We tested the performance of RGIS and OWLIM-SE on our LUBM Datasets. For a fair comparison between RGIS and OWLIM, both the systems used 2GB heap space and 3GB heap space when evaluating results for Data Set 1 and Data Set 2 respectively. Below is the execution time required for RGIS and OWLIM-SE for each of the datasets on each of the LUBM Test Queries.

LUBM Query	Data Set 1		Data Set 2	
	RGIS (In Sec.)	OWLIM-SE (In Sec.)	RGIS (In Sec.)	OWLIM-SE (In Sec.)
1	7.05	0.80	13.85	3.71
2	6.16	229.15	14.82	442.84
3	6.83	0.07	13.94	0.16
4	17.98	1.72	52.02	0.51
5	7.74	0.73	13.49	1.93
6	9.48	Could Not Load	21.46	Could Not Load
7	14.37	0.69	32.21	0.31
8	21.90	19.22	95.38	55.842
9	15.94	419.61	31.574	875.86
10	7.07	0.04	13.68	0.15
11	0.46	0.48	0.66	0.66
12	0.66	0.29	1.04	0.36
13	5.41	9.40	9.28	58.53
14	6.78	2170.36	27.00	Could Not Load

Table 15: Comparison of RGIS and OWLIM-SE

6.8 COMPARISON WITH MULGARA:

We compared the performance of RGIS with Mulgara on Data Set 1. For a fair comparison, we used 2GB heap space on both the systems when evaluating the LUBM Test queries on the Data Set. Below is the execution time required for RGIS and Mulgara for each of the datasets on each of the LUBM Test Queries.

LUBM Query	RGIS (In Sec.)	Mulgara (In Sec.)
1	7.05	1.69
2	6.16	29.115
3	6.83	1.77
4	17.98	15.52
5	7.74	0.36
6	9.48	0.13
7	14.37	0.99
8	21.90	0.91
9	15.94	45.339
10	7.07	0.03
11	0.46	0.45
12	0.66	0.24
13	5.41	2.83
14	6.78	0.11

Table 16: Execution Time of RGIS and Mulgara

We also compared the space required by Mulgara to store the LUBM Data Set. Below is the table that compares the storage space required by Mulgara and RGIS to store Data Set 1.

Data Set	Original Size	RGIS	Mulgara
Data Set 1	3.42 GB	1.14 GB	7.83 GB

Table 17: Storage Space required for RGIS and Mulgara

6.9 DISCUSSION

Execution Time:

- As seen from the results in table 11, the total execution time for the LUBM queries was six times than the execution time taken by RGIS.
- From the comparison of RGIS with Jena TDB on both the datasets, in most of the queries, RGIS performs much better than Jena TDB. Thus, from the results, we can safely say that we Jena TDB take at least twice the execution time than RGIS.
- Queries 2, 4, 7, 8, 9, all had large datasets and had more join operations involved when evaluating the query. For such queries, as seen from the results, RGIS performed better than Jena TDB and was 6 times faster than Jena TDB.
- For queries 1 & 3, the performance of Jena TDB was better than RGIS. We believe that this is because of the architecture it uses to save information about the Subject, Object and Predicates. Thus, queries that involve a single class, Jena TDB perform better because of the architecture of Jena TDB.
- Performance of RGIS was better than OWLIM-SE that involved data-intensive queries and for queries that required more complex joins.
- Performance of RGIS was better then Mulgara that required complex joins.

Storage Space:

- By using the custom storage format, we were able to reduce to size of the files by 66% that results into using less storage space.
- In comparison with Jena TDB, RGIS used $1/6^{th}$ the space required by Jena TDB to store the RDF/OWL datasets.
- The storage space required for Mulgara was 7 times more than that required for RGIS.

Heap Size/Memory:

- For Dataset 1, at least 2GB of heap memory is needed to evaluate and answer all the test queries.
- For Dataset 2, at least 3GB of heap memory is needed to evaluate and answer all the test queries.

CHAPTER 7

RELATED WORK

In recent years, many researchers have come up with different solutions to improve the efficiency of RDF Engines. Researchers have also proposed new architectures, techniques and models to store RDF data and to retrieve it. Previous experience and research has shown that one of the important reasons for low performance of a RDF engines is because of low performance on 'Join Operations' in SPARQL query processing. Hence, to address this problem, researchers have proposed using heuristics in SPARQL query planning [7][11][12] and rewriting query to improve efficiency [10].

Previous work has also suggested that using different storage technique can also help in improving performance of RDF engines. As described by researchers *Thomas Neumann and Gerhard Weikum* in their work [1], the authors replace the 'literals' with 'id' using a mapping directory. By using this approach, the authors argue, that the triples are compressed as we use 'id' instead of the 'literal'. Thus, this also helps in faster lookups and simple processing of a query. Thus, the authors say that we could use 'id' of literals when processing the query and once the query is processed, we can perform a simple lookup in the dictionary indexes to get back the literals. Besides, the authors have used 6 different tables i.e., all 6 different permutations of Subject, Object & Predicate, to store the different permutations represented by a triple in each of these tables, respectively. This is done so that it would result in faster lookup

when answering all different patterns of variables with variable in any position of the triple. The authors argue that they can afford to use 6 different tables to store the data because instead of storing the whole literal, they just store the 'id' of the literal. Hence, this level of redundancy is acceptable because it uses 'id' instead of literals and thus even with redundancy of data, they were able to still reduce the size of the dataset in the experiments.

In their work [2], authors *Javier D. Fernandez, Miguel A. Martinez-Prieto and Claudio Gutierrez*, talk about a new format to represent the data. The authors say that for efficient management of large RDF data, we can use the structural properties of RDF and split the data into three major components: Header, Dictionary & Triples. In this paper, the authors say that the header section can be used to store the meta-data information of the RDF graph. Further, in the dictionary section, we would be storing the literals and assigning them 'id'. In the triple section, instead of storing the literals, we can use their respective id mentioned in the dictionary section. Hence, the dictionary section would act as a lookup table for the '*triple'* section.

In RGIS, we give an index to every predicate, class attribute and property. Thus, for a literal, an index is stored in the class files and this index value is then referenced in the Property files. Thus, for every predicate, we just have single file and not 6 different files or internal structures as mentioned in RDF 3X. Besides, only the Property Files have a Header and Body. In the header, we store the indices of classes of both, subject and object.

Hence, by using a single file for every predicate and storing the data in Header-Body format, we do not need to store the data into 6 different formats. Besides, the Class Files can then be used to get the literal for a particular index. Hence, the advantage of using this format is that we load only those entries into the memory that are required instead of loading the entire dictionary

section as described in paper [2]. Hence, by loading only the required entries, RGIS utilizes less memory, as seen in the section 4.2.

Authors Mohammad Farhan Hussain, et al., in their work [4][6], discuss about a new format to store the RDF graph. In the paper, the authors say that they would first split the RDF data according to the Predicates, called as Predicate Split. Thus creating a file for every predicate in the Ontology. Further, the authors also split the RDF data based on implicit and explicit *'type information'* of the object, which the authors call as 'Predicate Object Split'. Thus, by using this custom format, the authors were able to reduce the storage space by 70%. Thus, to get the RDF data into the format discussed in the paper, we need to follow the steps below:

- Convert RDF data to N-Triples
- Predicate Split is then Applied to N-Triples data
- Predicate Object Split is then applied on the data received from above step.

In RGIS, we convert the data to Notation3 format and then split the data based on predicates and classes. Further, RGIS uses indexes for every Class and Class Objects, which is then referenced in the Property Files. Besides, we also assign index to every Property File. We also use *'Header-Body'* format to store information in the Property Files. Hence, by using this format, our approach gets distinguished from the approach mentioned in the papers above.

Further in [6], the authors describe an algorithm to evaluate queries such that they could be processed using Hadoop. In the algorithm, the authors try to find all the triples that are independent and are grouped together. Then, these triples are removed from the query and the algorithm is repeated on the rest of the remaining query to find the next independent triples. This is repeated till we exhaust all the triples that are asked in the query. Thus, the number of

repetitions would give us the number of jobs that are required on Hadoop and the group of triples obtained in every iteration can be used in the respective job on Hadoop.

Further, in [5], the authors say that they grouped independent triples into a job. Also, the performance of Hadoop is dependent on the number of Jobs that are required to evaluate a query. In the experiments, the authors have compared the performance on 2-Job Plan and 3-Job Plan. The experiments show that the performance of 2-Job Plan was better than the performance of 3-Job Plan. The reason for 2-Job Plan performing better was because of less read/write IO and less data transfer across the network.

In our approach, we create nodes for every variable that is encountered in the query. Then, based on the algorithm discussed in section 3.2, we generate a query plan that needs to be executed. Thus, there is a fundamental difference in both the approaches. The authors believe in grouping all triples irrespective of the variables and evaluating it. Whereas, in our approach, we would evaluate variables and the Object Properties associated with it. We believe that using our approach would result in better organization of data when evaluating complex queries that have more join operations. Our approach is compatible with Hadoop, but beyond scope of this Thesis.

Authors Lei Zou, et al., in their work [13], describe a novel indexing technique and pruning rules that help to reduce the query processing time. Further, the authors say that their technique can also be used to support wildcards in SPARQL query. In their technique, the authors have used adjacency list to store the RDF data. They then transform the RDF graph into signature graph by encoding all entities and vertex. They also use a novel indexing technique, VS*-tree, on this signature graph. The authors have also proposed filtering rules that can be used on the signature

graph to evaluate SPARQL queries. The same rules can also be used for SPARQL queries with wild cards.

Author Richard Cyganiak, in his work [3], gives us a brief overview of the relational algebra for SPARQL. In the paper, the author describes the techniques that can be used to evaluate the SPARQL query and convert it into equivalent SQL query. The author also gives an insight on how the Join operation of SPARQL can be mapped into SQL language.

In RGIS, we do not use the algebra or conversion to SQL mappings. However the algebra and techniques mentioned by the author were useful to us to perform the join operations and to evaluate the query. In RGIS, when we generate the query plan, we store the Meta-data i.e., information about different files from which we need to load the data into the memory and perform operations to evaluate the query. Thus, the techniques mentioned in the paper were helpful to us to determine the Meta-data required evaluating the queries.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 CONCLUSION

Performance and Storage technique are the important areas of research for RDF engines. In RGIS, we address each of these problems and provide solutions to them. Using its custom data and indexing format, RGIS uses less storage space to store RDF data. The intelligent Query Planner and Executor evaluate the SPARQL query and selectively load the files into memory and hence utilize less memory. Further, as the data is split based on Classes and Object Properties, we are able to reduce the search space. Thus, by selectively loading the files into memory, we load only the triples that are required to evaluate the query. Hence, RGIS helps reduce memory overhead.

RGIS has been tested on a dataset that has 83Million triples. RGIS not only used less storage space to store RDF data than Jena TDB but also executed the SPARQL queries faster than Jena TDB. For complex queries that involved complex join operations, performance of RGIS was faster than Jena TDB and OWLIM. Performance of RGIS was better than Mulgara for few of LUBM test queries. Hence, as suggested from the results, our algorithm performs better for SPARQL queries that involves complex join operations.

8.2 Future Work

RGIS stores the class hierarchy in *Class Hierarchy File*. We are currently working to create a parser that would parse the RDF data and generate the Class hierarchy data and store it in the Class Hierarchy File.

Currently, RGIS can evaluate SPARQL queries to find Subject and/or Object. We are working on RGIS so that it would also evaluate SPARQL queries that require finding relations between Subject-Objects. Thus, RGIS would also answer queries that have predicates as unknowns/variables. We would also be working to add the different features that are supported by SPARQL. E.g.: ORDERBY, FILTER, etc.

To improve the performance of RGIS, we would also like to implement a cache, which would cache the results/ data and thus improve the response time for the queries.

Since the Query Plan and Custom Format to store the data are generic, they could also be used on a distributed framework like Hadoop. Thus, in the future, we would also be working to test RGIS on a Hadoop. We believe that by using Hadoop, we would be able to support a bigger dataset.

REFERENCES

[1] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style Engine for RDF. In *Proceedings of the VLDB Endowment, Volume 1*, Pages 647-659, 2008.

[2] Fernández, Javier, Miguel Martínez-Prieto, and Claudio Gutierrez. "Compact representation of large RDF data sets for publishing and exchange." *The Semantic Web–ISWC 2010* (2010): 193-208.

[3] Cyganiak, Richard. "A relational algebra for SPARQL." *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170* (2005).

[4] Husain, Mohammad Farhan, et al. "Data intensive query processing for large RDF graphs using cloud computing tools." *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010.

[5] Husain, Mohammad, et al. "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing." *Knowledge and Data Engineering, IEEE Transactions on* 23.9 (2011): 1312-1327.

[6] Farhan Husain, Mohammad, et al. "Storage and retrieval of large rdf graph using hadoop and mapreduce." *Cloud Computing* (2009): 680-686.

[7] Stocker, Markus, et al. "SPARQL basic graph pattern optimization using selectivity estimation." *Proceedings of the 17th international conference on World Wide Web*. ACM, 2008.

[8] Huang, Jiewen, Daniel J. Abadi, and Kun Ren. "Scalable sparql querying of large rdf graphs." *Proceedings of the VLDB Endowment* 4.11 (2011).

[9] Myung, Jaeseok, Jongheum Yeon, and Sang-goo Lee. "SPARQL basic graph pattern processing with iterative MapReduce." *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*. ACM, 2010.

[10] Hartig, Olaf, and Ralf Heese. "The SPARQL query graph model for query optimization." *The Semantic Web: Research and Applications* (2007): 564-578.

[11] Groppe, Sven, Jinghua Groppe, and Volker Linnemann. "Using an Index of Precomputed Joins in order to speed up SPARQL Processing." *Proceedings 9th International Conference on Enterprise Information Systems (ICEIS 2007 (1), Volume DISI), Funchal, Madeira, Portugal, INSTICC (June 12-16 2007)*. 2007.

[12] Neumann, Thomas, and Gerhard Weikum. "Scalable join processing on very large RDF graphs." *Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, 2009.

[13] Zou, Lei, et al. "gStore: answering SPARQL queries via subgraph matching."*Proceedings of the VLDB Endowment* 4.8 (2011): 482-493.

[14] Notation3. http://en.wikipedia.org/wiki/Notation_3

[15] Notation3. http://www.w3.org/TeamSubmission/n3/

[16] RDF. http://www.w3.org/RDF/

[17] RDF. http://en.wikipedia.org/wiki/Resource_Description_Framework

[18] SPARQL. http://en.wikipedia.org/wiki/SPARQL

[19] SPARQL. http://www.w3.org/TR/rdf-sparql-query/

[20] Jena TDB. http://jena.apache.org/documentation/tdb/index.html

[21] Jena TDB Architecture. http://jena.apache.org/documentation/tdb/architecture.html

[22] Jena TDB and other RDF engines.http://www.w3.org/wiki/LargeTripleStores

[23] Performance of OWLIM and Jena TDB.http://www.ontotext.com/owlim/benchmark-results/owlim-jena-performance

[24] Jena SDB. http://jena.apache.org/documentation/sdb/query_performance.html

[25] LUBM. http://swat.cse.lehigh.edu/projects/lubm/

[26] LUBM Test Queries. http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt

[27] Jena. http://jena.apache.org/

[28] OWLIM. http://www.ontotext.com/owlim

[29] Sesame. http://www.openrdf.org/

[30] RDF-3X. http://www.mpi-inf.mpg.de/~neumann/rdf3x/

[31] http://en.wikipedia.org/wiki/Turtle_(syntax)