A NEW FILTERING INDEX FOR FAST PROCESSING OF SPARQL QUERIES

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

SRIVENU PATURI

Bachelor of Technology, Jawaharlal Nehru Technological University, 2010

Kansas City, Missouri

2013

A FILTERING INDEX FOR EFFICIENT RDF QUERY PROCESSING

Srivenu Paturi, Candidate for the Master of Science Degree

University of Missouri-Kansas City, 2013

ABSTRACT

The Resource Description Framework (RDF) has become a popular data model for representing data on the Web. Using RDF, any assertion can be represented as a (subject, predicate, object) triple. Essentially, RDF datasets can be viewed as directed, labeled graphs. Queries on RDF data are written using the SPARQL query language and contain basic graph patterns (BGPs). We present a new filtering index and query processing technique for processing large BGPs in SPARQL queries. Our approach called RIS treats RDF graphs as "first-class citizens." Unlike previous scalable approaches that store RDF data as triples in an RDBMS and process SPARQL queries by executing appropriate SQL queries, RIS aims to speed up query processing by reducing the processing cost of join operations. In RIS, RDF graphs are mapped into signatures, which are multisets. These signatures are grouped based on a similarity metric and indexed using Counting Bloom Filters. During query processing, the Counting Bloom Filters are checked to filter out non-matches, and finally the candidates are verified using Apache Jena. The filtering step prunes away a large portion of the dataset and results in faster processing of queries. We have conducted an in-depth performance evaluation using the Lehigh University Benchmark (LUBM) dataset and SPARQL queries containing large BGPs. We compared RIS with RDF-3X, which is a state-of-the-art scalable RDF querying engine that uses an

RDBMS. RIS can significantly outperform RDF-3X in terms of total execution time for the tested dataset and queries.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a thesis titled "A Filtering Index For Efficient RDF Query Processing," presented by Srivenu Paturi, candidate for the Master of Science degree, and certify that in their opinion, it is worthy of acceptance.

Supervisory Committee

Praveen Rao, Ph.D., Committee Chair
School of Computing and Engineering

Yugyung Lee, Ph.D.
School of Computing and Engineering

Lein Harn, Ph.D.
School of Computing and Engineering

TABLE OF CONTENTS

ABSTRACT iv

TABLE OF CONTENTS vii

ILLUSTRATIONS x

ACKNOWLEDGEMENTS xiv

CHAPTER

1 INTRODUCTION 1

1.1 Focus of the Thesis 2

2 MOTIVATION AND RELATED WORK 4

2.1 Motivation 4

3 BACKGROUND 6

3.1 Introduction 6

3.2 A Triple in RDF 6

3.3 More Complex Examples 7

3.4 Sample Data 7

3.5 Sample Query 8

3.6 Results for the Sample Query 9

4 BLOOM FILTERS AND BLOOM COUNTERS 10

4.1 Bloom Filters 10

4.2 Bloom Filter Generation 10

4.3 An Example of Bloom Filter Generation 11

4.3 An Example of Bloom Comparison 11

4.5 Limitations of Bloom Filters and Multi-sets 15

4.6 Counting Bloom Filters 18

4.7 Counter Comparison 18

4.8 An Example of Counting Bloom Filters Generation 19

4.9 An Example of Counting Bloom Filters Comparison.......... 20

4.10 Bloom Parameters and false positives 21

5 SYSTEM ARCHITECTURE 22

6 SIGNATURE TREE GENERATION 24

6.1 Signature Trees 24

6.2 Grouping of Signatures 24

6.3 Tree Construction.......... 25

6.4 An Example of our Signature Tree Construction 26

6.4.1 Processing all sets (Recursion level 0) 26

6.4.2 Begin Processing Group A (Recursion level 1).......... 28

6.4.3 Begin Processing Group A.A (Recursion level 2).......... 30

6.4.4 End Processing Group A.A 32

6.4.5 Begin Processing Group A.B (Recursion level 2).......... 32

6.4.6 End Processing Group A.B.......... 33

6.4.7 Begin Processing Group B (Recursion level 1).......... 33

6.4.8 End Processing Group B.......... 35

6.4.9 End Processing all Groups.......... 35

6.4.10 All leaf Nodes in the tree.......... 35

6.5 Construction of Bloom Counters for Each Node in the tree.......... 36

7 QUERYING THE SIGNATURE TREE 37

7.1 Our Querying Approach 37

7.2 Candidate Matches 38

7.3 Query Classification 38

7.4 Low Selectivity Query 38

7.5 High Selectivity Queries 39

7.8 High Selectivity Querying Challenges 40

7.9 Sample Queries 41

8 EVALUATION 46

8.1 Implementation 46

8.2 Supplementary Tools Used 46

9 EXPERIMENTS 47

9.1 Experiment Set 1 – Varying Fanout Size 47

9.2 Experiment Set 2 – Varying Seed Pick Size 62

10 CONCLUSION 77

REFERENCES 78

VITA 83

ILLUSTRATIONS

Figure Page

3.1. A Graph Representation of RDF Triple Entities... 6

3.2. A Simple Example in RDF Graph Representation. 7

3.3. A Comprehensive Example in RDF Graph Representation. 8

3.4. A Query for the above RDF Graph.. 9

4.1. Initialized Data Bloom Filter. 11

4.2. Inserting First Element into the Data Bloom Filter. 12

4.3. Inserting Second Element into the Data Bloom Filter. 12

4.4. Fully Built Data Bloom Filter. 13

4.5. Initialized Query Bloom Filter. 14

4.6. Inserting Elements into the Query Bloom Filter. 14

4.7. Fully Built Query Bloom Filter. 14

4.8. Comparison Between Data Bloom Filter and Query Bloom Filter.......................... 15

4.9. Initialization of Bloom Filter For Multi-Sets S and Q. 16

4.10. A Bloom Filter Generated with Data Multi-Set S. 17

4.11. A Bloom Filter Generated with Query Multi-Set Q. 17

4.12. Comparison Between Multi-Set S and Multi-Set Q Bloom Filters. 17

4.13. Initialization of a Counting Bloom Filter With 2 Bit Counter Size. 19

4.14. Counting Bloom Filter Generated with data Multi-set S. 20

4.15. Counting Bloom Filter Generated with query Multi-set Q. 20

4.16. Comparing Counting Bloom Filters. 20

5.1. RIS Indexing Architecture. 22

5.2. RIS Querying Architecture. 23

6.1. ‘K’ Highest Cardinality Signatures in the Signature Set. 27

6.2 Distribution of Signatures at the Recursion Level 0. 27

6.3. Fanout Size Checking for Each Group at Recursion Level 0. 28

6.4. ‘K’ Highest Cardinality Signatures in Group A. 29

6.5 Distribution of Signatures at the Recursion Level 1. 29

6.6. Fanout Size Checking for Each Group at Recursion Level 1. 30

6.7. ‘K’ Highest Cardinality Signatures in Group A.A.. 30

6.8. Distribution of Signatures at the Recursion Level 2. 31

6.9. Fanout Size Checking for Each Group at Recursion Level 2. 32

6.10. Constructing Leaf Node 2. 32

6.11. ‘K’ Highest Cardinality Signatures in Group B. 33

6.12. Distribution of Signatures at the Recursion Level 1. 34

6.13. Fanout Size Checking for Each Group at Recursion Level 1. 34

6.14. Showing all Finalized Leaf Nodes LN0, LN1, LN2, LN3, LN4. 35

6.15. Construction of Bloom Counters for Each Node. 36

TABLES

Table Page

9.1.1 Indexing Evaluation for Fanout as 250. 48

9.1.2 Querying Evaluation for Fanout as 250. 49

9.1.3 Performance Comparison for Fanout as 250. 50

9.1.4 Indexing Evaluation for Fanout as 500. 50

9.1.5 Querying Evaluation for Fanout as 500. 52

9.1.6 Performance Comparison for Fanout as 500. 53

9.1.7 Indexing Evaluation for Fanout as 1000. 53

9.1.8 Querying Evaluation for Fanout as 1000. 54

9.1.9 Performance Comparison for Fanout as 1000. 55

9.1.10 Indexing Evaluation for Fanout as 2000. 56

9.1.11 Querying Evaluation for Fanout as 2000. 57

9.1.12 Performance Comparison for Fanout as 2000. 58

9.2.1 Indexing Evaluation for Seed Pick Size as 100. 62

9.2.2 Querying Evaluation for Seed Pick Size as 100. 64

9.2.3 Performance Comparison for Seed Pick Size as 100. 65

9.2.4 Indexing Evaluation for Seed Pick Size as 200. 65

9.2.5 Querying Evaluation for Seed Pick Size as 200. 66

9.2.6 Performance Comparison for Seed Pick Size as 200. 67

9.2.7 Indexing Evaluation for Seed Pick Size as 400. 68

9.2.8 Querying Evaluation for Seed Pick Size as 400. 69

9.2.9 Performance Comparison for Seed Pick Size as 400. 70

9.2.10 Indexing Evaluation for Seed Pick Size as 800. 71

9.2.11 Querying Evaluation for Seed Pick Size as 800. 72

9.2.12 Performance Comparison for Seed Pick Size as 800. 73

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my thesis advisor Dr. Praveen Rao. He has been a phenomenal mentor and teacher to work with. He played a very significant role with his valuable guidance and outlook which helped me shape better as a software engineer. I should also thank him for providing me with all the financial support, lab equipment and excellent work atmosphere which let me be very productive during my research work. I am also thankful to National Science Foundation for supporting this work under Grant No. 1115871.

Although many helped me in the development of this tool, I would like to specifically thank Vasil Slavov for his help in integrating Apache JenaTDB with our proposed RIS tool and Dinesh Barenkala for providing us with the input graph datasets.

I am also grateful to my family for their constant support during my graduate program. Last but not the least, I also thank my peers in the research lab for all the knowledge and fun that was shared.

CHAPTER 1

INTRODUCTION

In recent times much research has been done on semi-structure data. One of the challenging problems that is being studied is to manage semi structure data. As the name suggests semi-structured data is a form of structured data that does not confirm with the formal structure of data models associated with the relational databases. One of the widely represented semi structure formats is RDF (Resource Description Framework) specification. RDF is a popular language for representing data on the Web.

The essence of RDF lies in the notion of representing any fact as subject, predicate, and object. Formally, RDF represents resources as a directed, labeled graph where a pair of adjacent nodes denotes two things and the directed, labeled edge represents their relationship. The source node denotes the subject; the sink node denotes the object; and the edge label is the predicate (or property). This "subject-predicate-object" relationship is commonly referred to as an RDF triple. SPARQL is a popular query language for RDF graphs. Using SPARQL, complex graph pattern queries can be expressed on individual RDF graphs as well as across multiple RDF graphs. In recent years, the RDF data model has become increasingly important in domain-specific applications and the WWW. Through RDF technologies, one can reason over semantic data, which is highly appealing in domains such as healthcare, defense and intelligence, biopharmaceuticals, and so forth. With the rapidly growing size of RDF datasets (e.g., DBPedia, Billion Triples Challenge), there is a pressing need for efficient RDF processing tools.

1.1 Focus of the Thesis

The focus of the thesis is to design, develop and evaluate an RDF indexing and querying technique with the following components –

- Indexing
  1. Signature Generation: Every triple in a connected component is processed by generating six other arrangements for it by swapping uri's with variables.
  2. Indexing Using Bloom Counters: Each signature is then stored in bloom counters. Storing signatures in bloom counters helps us use perform quick bitwise operations. This we plan to use at the time of querying to quickly eliminate non-potential signatures from querying without actually going through the entire data set, thus saving a lot of time.
  3. Filtering Index Generation: All the Bloom counters for each signature from the above step are stored in a special signature tree for quick traversals and retrievals.
- Querying
  1. Query Signature Generation: Just like the Data Signature, Query Signature is generated by running the query through signature generator.
  2. Query Bloom Counters: A bloom counter is constructed for the query signature.
  3. Candidate Matches: By comparing the query bloom counter with data set bloom counters, potential candidates are identified which may contain the

results.

4. Query Execution: In the final step, RDF-3X tool is run only on the candidate signatures to produce the results.

- Performance Evaluation

  1. An extensive performance evaluation for this approach has been done using some of the popular RDF datasets. We choose LUBM, which is a synthetic RDF dataset to conduct our experiments.
  2. The queries for LUBM data can be classified into two sets - Queries with large number of joins and queries with less number of joins. We show that our approach outclasses the existing methods in the case of queries with large number of joins while in the case of queries with lesser joins, our approach is quite comparable with the existing methods.

The rest of the document is organized as follows. Chapter 2 discusses the background and motivation while Chapter 3 discusses Bloom Counter Generation and Comparison. Chapter 4 discusses our architecture. Chapter 5 discusses our Index Generation. Chapter 5 discusses our Query Processing. Chapter 7 discusses our evaluation. Finally, we conclude in chapter 8.

CHAPTER 2
MOTIVATION AND RELATED WORK

2.1 Motivation

Today, there are a number of open-source and commercial tools for storing and querying RDF graphs. These tools either store and process RDF in main-memory, use an RDBMS, or a native RDF database. The popular approach has been to use relational database systems for storing, indexing, and querying RDF. Abadi et al. proposed a vertical partitioning approach and leveraged a column-oriented DBMS for achieving an order of magnitude performance improvement over previous techniques [16]. RDF-3X [17] and Hexastore [18] demonstrated that storing RDF data in a single triples table and building exhaustive indexes on the six per mutations of (s, p, o) triples can significantly outperform the vertical partitioning approach [16] and also support a larger class of RDF queries efficiently. Recently, BitMat [19] was proposed to overcome the overhead of large intermediate join results in RDF-3X and Hexastore when queries contain low selectivity triple patterns. (Low selectivity implies large result set size.) There are some RDF stores that operate in shared-nothing clusters (e.g., YARS2 [20], 4store [21], Clustered TBD [22]) by hashing triples/quadruples and distributing them on different nodes in the cluster. Parallel query processing is performed. The scalability of these approaches has been demonstrated on small sized clusters. Weaver et al. [23] have studied RDF query processing on supercomputers. More recently, tools for data intensive computing such as Apache Hadoop and Pig have been used for query processing and analytics over RDF data [24], [25], [26]. These approaches are more suitable for batch processing of queries. A few researchers have focused on parallel RDF reasoning [27], [28]. More recently, Huang et al. , developed parallel RDF query processing techniques for large RDF graphs [29].

CHAPTER 3

BACKGROUND

3.1 Introduction

RDF stands for Resource Description Framework. It is an idea to represent statements regarding a resource as an expression involving subject, predicate and object. In RDF terminology, these expressions are often referred to as RDF triples. In an RDF triple, a subject is the resource involved in the statement, predicate is a property about the resource and object is a defining characteristic of the source connected via a predicate. In most cases, predicates represent a relationship between the subject and object. Usually, a dot '.' Is used at the end of a triple to mark that the expression is complete.

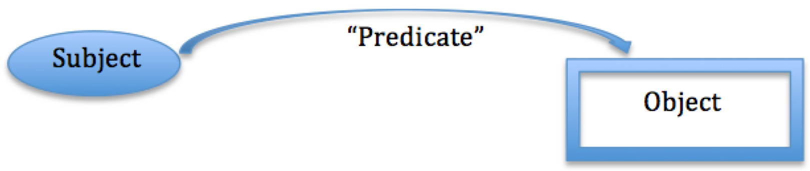


Figure 3.1. A Graph Representation of RDF Triple Entities.

3.2 A Triple in RDF

Let as consider an example statement "**Bob studies at school"**. Now, let us represent this piece of data in the RDF format. In the RDF format -

Subject/Resource is "Bob",

Predicate/property is “studies at” and,

Object is “school”

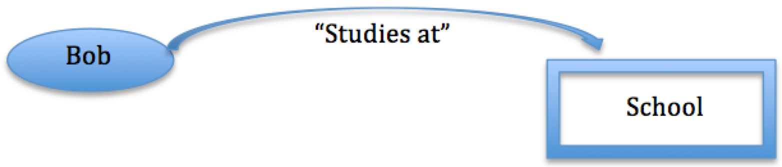


Figure 3.2. A Simple Example in RDF Graph Representation.

3.3 More Complex Examples

RDF graphs can be used to illustrate complex graphs too. There could be graphs that describe millions of subjects and have millions of properties and objects.

It is also very typical that in these complex graphs an object in one triple acts as a subject for another. For instance, in the above simple example we can extend the graphs with statements like ‘School has name xyz’ or ‘School has teachers’. In the next few topics we show how to visualize a comprehensive data and how to query it.

3.4 Sample Data

Below is a more complex RDF graph that has multiple subjects and objects inter-connected via different predicates-

<Book> <title> <Lean Ruby> .

<Book> <sn> <100007> .

<Book> <Publisher> <Techno Press> .

<Techno Press> <Phone> <530-XXX-XXXX> .

<Book> <Author> <Bob> .

<Bob> <Phone> <816-XXX-XXXX> .

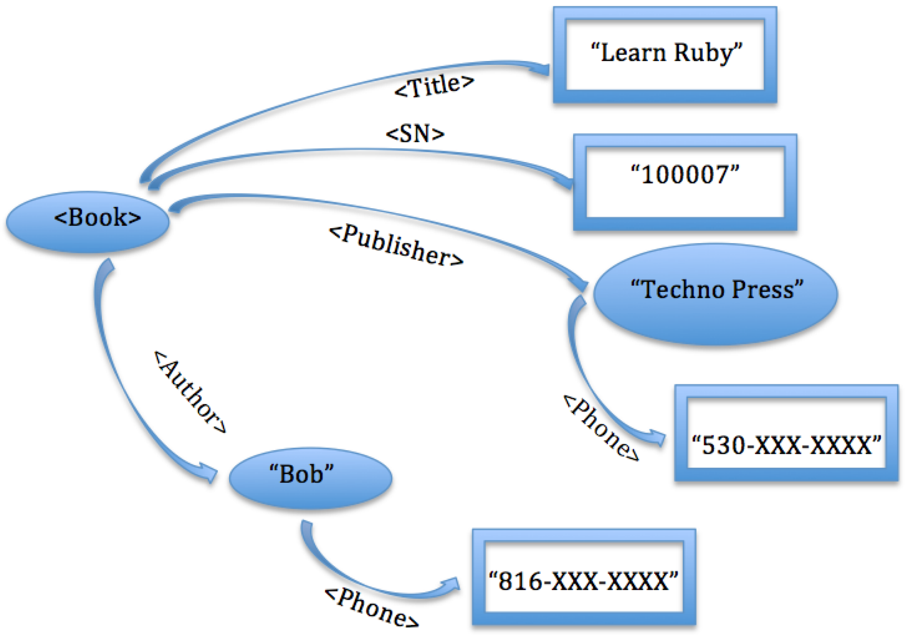


Figure 3.3. A Comprehensive Example in RDF Graph Representation.

3.5 Sample Query

Below is a sample query for the above data. The query has just one triple and two variables. The triple is intended to find all the phone numbers from the data and prints the name and number records accordingly.

SELECT ?name, ?number WHERE

{

```
  ?name <phone> ?number
}
```

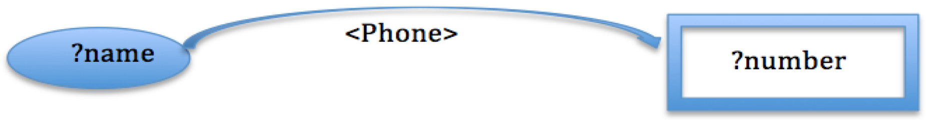


Figure 3.4. A Query for the above RDF Graph.

3.6 Results for the Sample Query

The above query for the given data would fetch two results. The values of these results would correspond to the variables (with a '?' prefix) selected in the select segment of the query. Following are the results –

<Techno Press> <530-XXX-XXXX>

<Bob> <816-XXX-XXXX>

The queries for RDF data can be more complex than this, specifically when they have a lot of joins involved. Most of the contemporary querying tools consume a lot of time to return results when the queries are large and have as many as 20 triples in the query. The time consumption to return results also depends on the number of variables involved and the number of self joins with the query. Usually the more number of self joins and variable, much longer the processing will take to return the results.

CHAPTER 4

BLOOM FILTERS AND BLOOM COUNTERS

4.1 Bloom Filters

A bloom filter is a memory efficient data structure to determine the probability of the presence of an element in a sample set. It can be used to test whether an element E is a member of a set S or not.

A bloom filter is implemented as a bit array of n bits where the element E's presence in the set is estimated by evaluating if the k^{th} bit where $(k < n)$ is set in the bloom filter or not.

There are two steps involved in the usage of bloom filters – Bloom Filter Generation and Bloom Filter Comparison. The Bloom Filter Generation deals with constructing the bloom filters and storing the computational results of the input data. The later deals with comparing the computed query with the built bloom filters from the first step. These processes are explained in detail in the next few topics.

4.2 Bloom Filter Generation

In this step, a bloom filter is constructed and its bits are set based on a few operations on the set S. There are many parameters involved in the construction of bloom filter of size N. One important parameter is forming K hash functions H1, H2, H3, . . . HK such that each of those hash functions return a value between 1 and N when applied on an element E.

Initially when a bloom filter with size N is constructed, all the bits in the bit array are set to zero. Then, the value R where R<N which is returned by applying hash function H on an element E represents the bit position that needs to be set in the bloom filter.

4.3 An Example of Bloom Filter Generation

In this topic we give an in depth explanation of how a bloom filter can be generated from given data using a comprehensive example. The various constants used in the below computations are picked randomly for the explanation purpose. Below are the parameters to construct a bloom filter B -

Let the set S to be inserted be - {3, 14, 19}

Let the size N of the bloom filter B be - 16

Let the number of hash functions be - 3

Let the hash function H1 be - ((5x + 4) % 20)%16

Let the hash function H2 be - ((9x + 10) % 20)%16

Let the hash function H3 be - ((2x + 11) % 20)%16

Below are the sequential steps of how a bloom filter is built –

1) Initial values in a bloom filter B with size 16

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.1. Initialized Data Bloom Filter.

2) Inserting the results of hash functions set on the first element 3

R1(3) = ((5*3 + 14) % 20)%16 = 9

R2(3) = ((9*3 + 10) % 20)%16 = 1

R2(3) = ((2*3 + 11) % 20)%16 = 1

0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.2. Inserting First Element into the Data Bloom Filter.

3) Inserting the results of hash functions set on the second element 14

R1(14) = ((5*14 + 14) % 20)%16 = 4

R2(14) = ((9*14 + 10) % 20)%16 = 0

R2(14) = ((2*14 + 11) % 20)%16 = 3

0	0	0	0	0	0	1	0	0	0	0	1	1	0	1	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.3. Inserting Second Element into the Data Bloom Filter.

4) Inserting the results of hash functions set on the third element 19

R1(19) = ((5*19 + 14) % 20)%16 = 3

R2(19) = ((9*19 + 10) % 20)%16 = 11

R2(19) = ((2*19 + 11) % 20)%16 = 9

0	0	0	0	1	0	1	0	0	0	0	1	1	0	1	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.4. Fully Built Data Bloom Filter.

4.4 An Example Bloom Filter Comparison

Once the bloom filter for set S is generated, bloom filter comparison operation is used to find out the possibility of a query set S2 to be a subset of set S. In this step, another bloom filter B2 is constructed out of the query set S2 using the same set of hash functions H1, H2, H3, . . . Hk used in the first operation.

After we construct the query bloom filter Q, we compare it with the data bloom filter B to check if the bits set in bloom filter Q are also set in bloom filter B. If they are set in B as well, then we say that we have a potential candidate match. Otherwise, if the bits set in Q are not set in B, we can safely assert that the query set S2 is not a part of the data set S.

Below is an example of how query set S2 is computed and compared to determine the possibility of it being a part of data set S -

Let the query set S2 be - {3, 11}

The size N of the bloom filter B is - 16

The number of hash functions is - 3

The hash function H1 is - ((5x + 4) % 20)%16

The hash function H2 is - ((9x + 10) % 20)%16

The hash function H3 is - ((2x + 11) % 20)%16

1) Initial values in a bloom filter B with size 16

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.5. Initialized Query Bloom Filter.

2) Inserting the results of hash functions set on the first element 3

R1(3) = ((5*3 + 14) % 20)%16 = 9

R2(3) = ((9*3 + 10) % 20)%16 = 1

R2(3) = ((2*3 + 11) % 20)%16 = 1

0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.6. Inserting Elements into the Query Bloom Filter.

3) Inserting the results of hash functions set on the second element 11

R1(11) = ((5*11 + 14) % 20)%16 = 3

R2(11) = ((9*11 + 10) % 20)%16 = 3

R2(11) = ((2*11 + 11) % 20)%16 = 13

0	0	1	0	0	0	1	0	0	0	0	0	1	0	1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.7. Fully Built Query Bloom Filter.

4) Comparing the query bloom filter Q with data bloom filter S

The comparison of the query bloom filter Q with data bloom filter S is show as below. For each bit from 0 to 15, the set bits in Q are being checked with S. If that bit is also set in S then the algorithm moves to the next set bit in Q otherwise, it is confirmed that the query returns false.

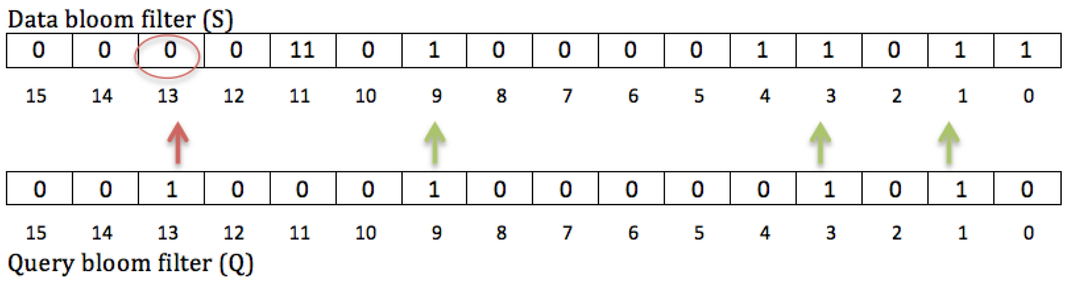


Figure 4.8. Comparison Between Data Bloom Filter and Query Bloom Filter.

In the above example, the values for set bit locations in 1, 3 & 9 were matched in both the query set Q and data set S. But there was a difference in the bit located 13 where it was set for query bloom filter Q but not in the data bloom filter S. Hence, it is confirmed that the query set S2 is not part of the data set S, which is true.

4.5 Limitations of Bloom Filters and Multi-sets

One of the main drawbacks using bloom filters is it can sometimes be impossible to distinguish between two different multi-sets using the same set of elements. In such cases, bloom filters return guaranteed false positives. A false positive is a result where bloom filters return a true value for a query but, when the query is run on the actual dataset, it returns no results. One common scenario where bloom filters return a false positive is when multi-set query is used on a multi-set data where the set elements of query is a subset of set elements of the multi-set data.

For example, below is a multi-set data and multi-set query, which use the same elements – 3,6,8

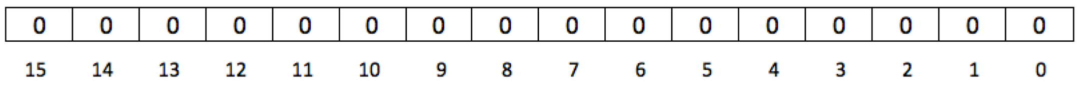
Data Multi-set S = {3, 3, 6 , 8, 8}

Query Multi-set Q = {3, 6, 6, 6}

In the above example, clearly the query multi-set Q is not a subset of data multi-set S because Q has three instances of element-8 where as S has only two. Let us try to apply the above sets on a simple bloom filter -

1) Bloom Filter Construction

Let the size N of the bloom filters for above multi-sets S and Q be - 16



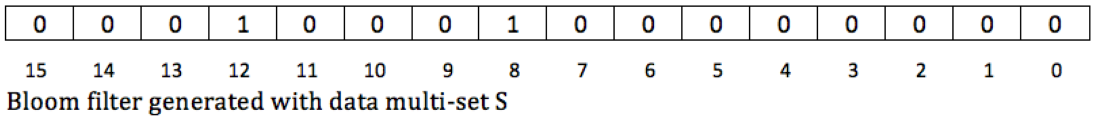
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.9. Initialization of Bloom Filter For Multi-Sets S and Q.

2) Bloom Filter for Data Multi-set S {3, 3, 6, 8, 8}

Let the number of hash functions be - 1

Let the hash function H1 be - ((8x + 4) % 20)%16



0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

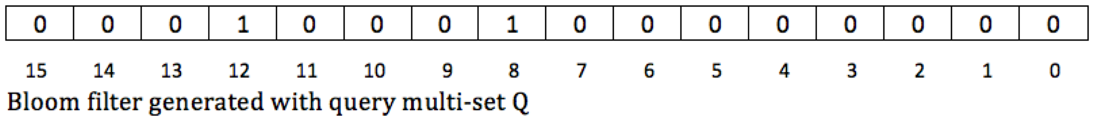
Bloom filter generated with data multi-set S

Figure 4.10. A Bloom Filter Generated with Data Multi-Set S.

3) Bloom Filter for Query Multi-set Q {3, 6, 6, 6}

Let the number of hash functions be - 1

Let the hash function H1 be - ((8x + 4) % 20)%16

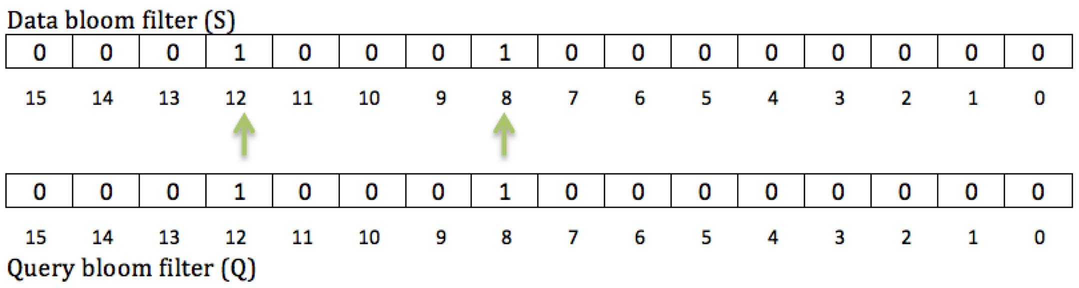


0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bloom filter generated with query multi-set Q

Figure 4.11. A Bloom Filter Generated with Query Multi-Set Q.

4) Comparing Query bloom filters for Multi-set Q with Data Multi-set S



Data bloom filter (S)

0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Query bloom filter (Q)

Figure 4.12. Comparison Between Multi-Set S and Multi-Set Q Bloom Filters.

The comparison returns true that Q is a subset of S while it is not. Therefore, this results in the potential candidate match to be a false positive. Hence, it is shown that bloom filters fail to operate affectively on multi-sets.

4.6 Counting Bloom Filters

A counting bloom filter is an extension of the regular bloom filter. It is also an array where bits can be set to determine the probability of existence of elements in a set. One of the major differences between counting bloom filters and bloom filters is counting bloom filters dedicate multiple bits in the array for each return value from the hash set where as bloom filters dedicate only one bit in the array for each return value from the hash set. Dedicating multiple bits in the array for each return value from the hash set enables counting bloom filters to count the number of times that value was returned for the whole data set until the counter is full. The values returned after the counter is full will overflow and can be ignored in the implementation.

The ability of counting the number of times the hash set has returned a value helps counting bloom filters to be more affective than the bloom filters and makes them functionally useful for even for multi-sets. For counting bloom filters to be affective against multi-sets the number of bits dedicated for each return value should be in proportion to the average number of repeated instances of an element in the multi-set.

4.7 Counter Comparison

During the comparison of query counting bloom filter with data counting bloom filter, the query multi-set is considered a subset of data multi-set as long as the counter value in the query filter is less than or equal to that of the data filter for each counter position. Then, the query would return true and the data multi-set is considered as a potential candidate for results.

If the counter value in the query filter is more than that of the data filter for at least one counter position, then it is asserted that the query multi-set cannot be a subset of data multi-set and the query returns false, thus confirming zero results.

4.8 An Example of Counting Bloom Filters Generation

Now, we show how counting bloom filters can overcome the limitations of bloom filters by using the above example where using a bloom filter resulted in a false positive.

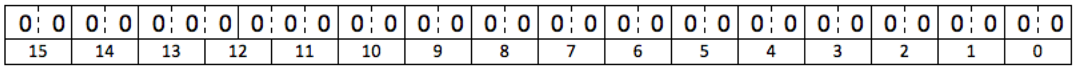
Data Multi-set S = {3, 3, 6, 8, 8}

Query Multi-set Q = {3, 6, 6, 6}

1) Counting Bloom Filter Construction

Let the size N of the bloom filter B be – 16

Let the counter size be 2 bits



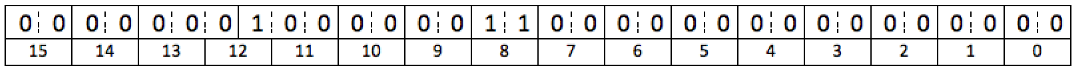
0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.13. Initialization of a Counting Bloom Filter With 2 Bit Counter Size.

2) Counting Bloom Filter for Data Multi-set S {3, 3, 6, 8, 8}

Let the number of hash functions be - 1

Let the hash function H1 be - ((8x + 4) % 20)%16



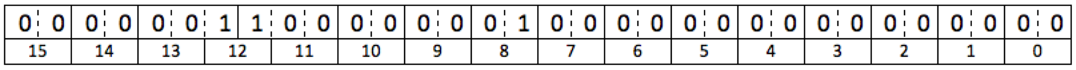
0 0	0 0	0 0	0 1	0 0	0 0	0 0	1 1	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.14. Counting Bloom Filter Generated with data Multi-set S.

3) Counting Bloom Filter for Query Multi-set Q {3, 6, 6, 6}

Let the number of hash functions be - 1

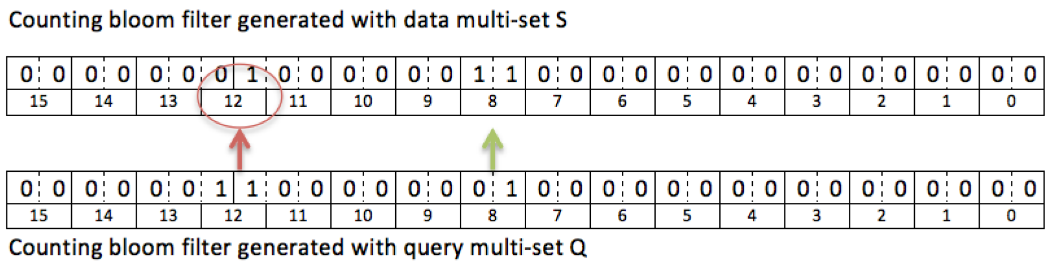
Let the hash function H1 be - ((8x + 4) % 20)%16



0 0	0 0	0 0	1 1	0 0	0 0	0 0	0 1	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.15. Counting Bloom Filter Generated with query Multi-set Q.

4.9 An Example of Counting Bloom Filters Comparison



Counting bloom filter generated with data multi-set S
Counting bloom filter generated with query multi-set Q

0 0	0 0	0 0	0 1	0 0	0 0	0 0	1 1	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0 0	0 0	0 0	1 1	0 0	0 0	0 0	0 1	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.16. Comparing Counting Bloom Filters for the Query Multi-set Q with Data Multi-set S.

In the above figure, the value in the 12th position in query filter is more than that of in data filter. It can be asserted that the query multi-set is not a subset of the data multi-set, therefore the query returns zero results.

4.10 Bloom Parameters and false positives

Bloom counter parameters have a large impact in determining the percentage of false positives during a bloom query comparison. The two significant parameters for bloom counters are the counter size and the number of bloom counters that are used for indexing. Having high values for both of these will ensure that the percentage of false positives will be low.

CHAPTER 5

SYSTEM ARCHITECHTURE

Our system, RDF Index via Signatures(RIS) uses the novelty of our approach as well as the advantages of existing RDF querying tools which are considered to be the benchmark standard.

The architecture consists of two main parts indexing and querying. Index contains components – one the graph identifying tool where we use bloom counters to identify which graph can have the potential result. One of the main purposes of this component is to eliminate the indexes which definitely do not yield any results and the other component is RDF-3X or JenaTdb. RDF-3X or JenaTdb are modern RDF querying tools. To use their services we index the data to their indexing formats.

In querying part we use RDF-3X in the case of small queries while we use JenaTdb for queries which are large.

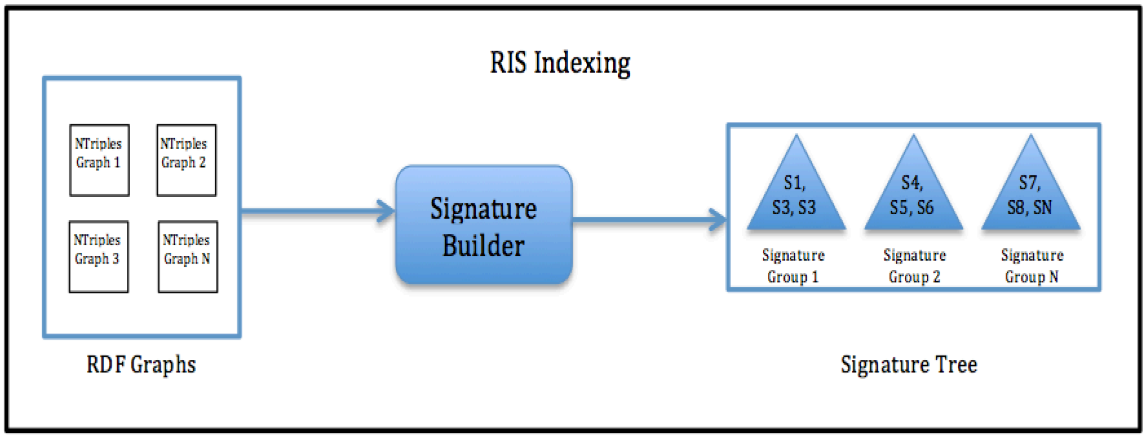


Figure 5.1. RIS Indexing Architecture.

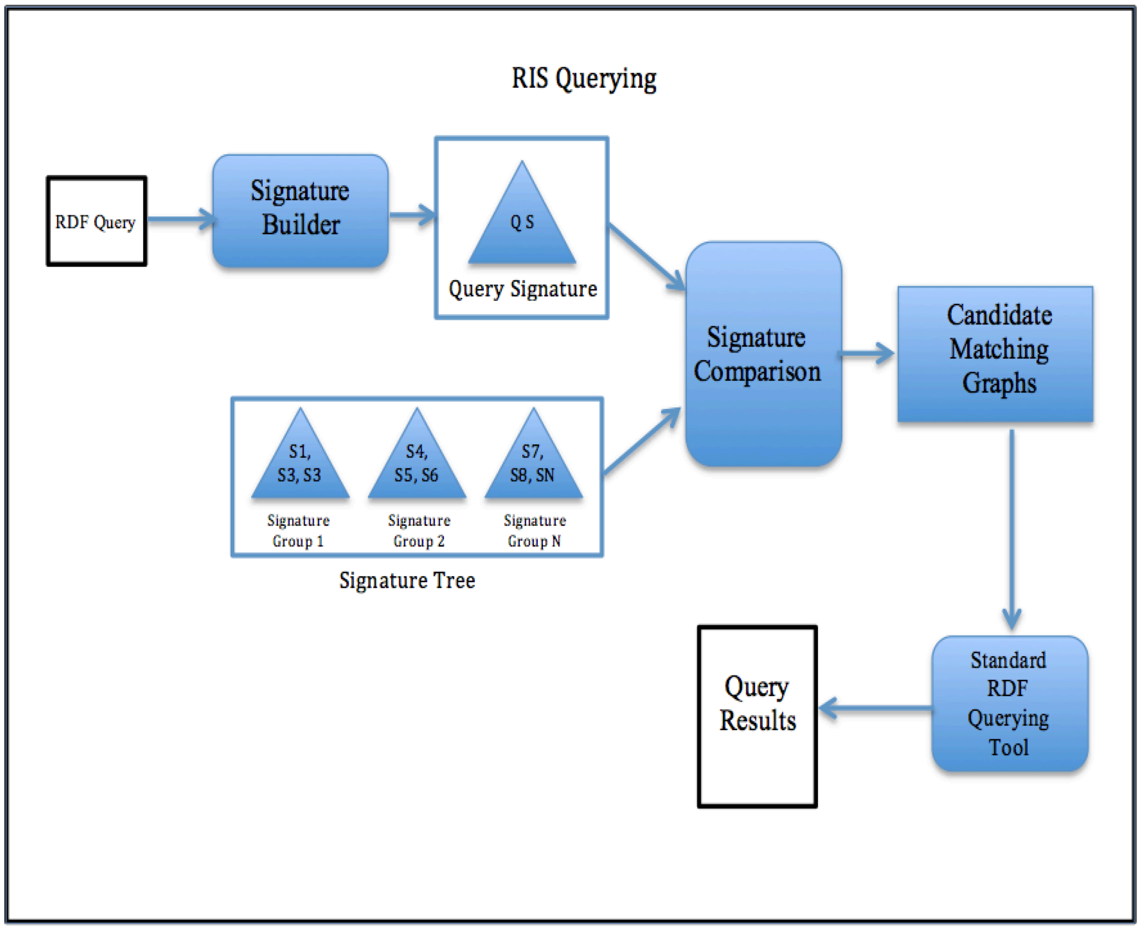


Figure 5.2. RIS Querying Architecture.

CHAPTER 6

SIGNATURE TREE GENERATION

6.1 Signature Trees

Once all the graphs are loaded in the bloom counters, it still is inefficient to run the query bloom comparison with all the bloom counters. Although the bloom comparison is done very fast due to the bit level operations involved, the process could consume a lot of time especially when the number of signatures is large. Often it is observed that there could be as many as hundreds of thousands of signatures in a large partitioned graph. In addition to this, we should also account the time it takes to run the actual query against the original graph partitions of candidate matches computed in the bloom comparison.

We address this problem by coming up with our own tree structure inspired by the GIS tool[1] . In this approach we recursively group a set of signatures based on their similarity.

6.2 Grouping of Signatures

The grouping of signatures is done based on the similarity between two signatures. The signatures that are more similar to each other tend to have more likelihood to end up in the same group. The similarity between two multi-sets can be calculated by the formula - $(S1 \cap S2) / (S1 \cup S2)$ where S1 and S2 are two multi-sets. Ideally, in this process we should compare each signature to every other signatures to find the best match to group it with. But, since that again reduces the efficiency of the system in the cases of large number of signatures, we use the concept of k highest cardinality signatures. Here we find two signatures among the k highest cardinality signatures from all the signatures. Then, we try to group rest of the signatures from those two groups. If the similarity of a signature is same with all the groups, we assign it to the group with least number of signatures in it.

We limit a group size by passing a parameter called fanout size. Fanout size can be defined as the maximum number of signatures a group can have. If a group exceeds the fanout size then, we recursively use the above process to split the group into two parts.

At the end of the grouping algorithm, we should be left with a few number of groups of signatures where each group contains those signatures that are the most similar to each other compared to the rest in the other groups.

6.3 Tree Construction

Once the grouping of signatures is done, we construct a tree from these groups in the bottom top approach. The tree is constructed such that the parent node will contain the union of all the elements in its child nodes. The intuition is that when traversing the tree at the time of querying, we can safely discard an entire sub tree if the root of that sub tree does not return any results. Since, the parent always contain the union of all the child nodes signatures, if a parent doesn't match a triplet constraint in the query, it can be guaranteed that none of the child elements also contain that triplet.

Once, the tree structure is formed, we construct bloom counters for each of the child nodes which contain the signature groups as explained above. Once the tree is traversed, should be able to identify the potentials candidates by discarding the unsatisfied sub trees there by running the query only on selected data signatures.

Such an approach helps us in successfully querying only a selected parts of the large data set in comparison with querying the entire data set which most of the currently used querying mechanisms follow. Thus one of our main goals to avoid querying all the data parts is achieved.

In our implementation we construct only a two level tree. This helps us to avoid long tree traversals, although often it is a trade-off between having long breadth versus having a big height in the tree.

6.4 An Example of our Signature Tree Construction

In this section, we will explain in detail about our signature tree construction with the help of an example. Below is the example to construct an index tree –

Let, A = {S0, S1, S2, S3, S4, S5, S6, S7, S8, S9} be the list of multi-set signatures to be processed.

Let, the fanout(fmax) parameter be 3

Let, the seed pick(k) size be 5

6.4.1 Processing all sets (Recursion level 0)

The algorithm is initiated by passing the parameters A = {S0, S1, S2, S3, S4, S5, S6, S7, S8, S9}, fanout and k.

The first step is to find the k highest Cardinality Signatures out of the set A. In this step, we select two signatures out of S0, S1, S2, S3, S4 such that the common elements between them is the least.

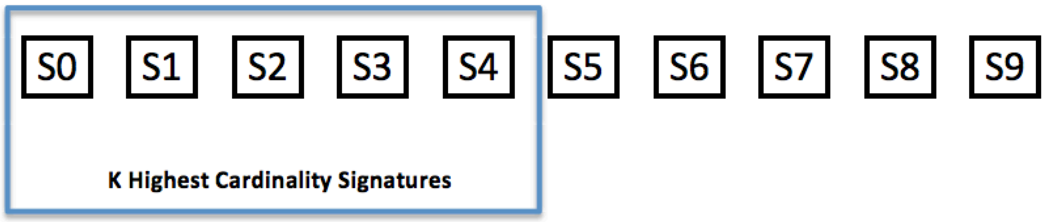


Figure 6.1. 'K' Highest Cardinality Signatures in the Signature Set.

Let S0 and S3 be the most dissimilar signatures found out. In this step we loop through rest of the signatures and club them with the group which they are most similar to using the formula - (Si∩Sj) / (Si∪Sj).

The below diagram shows how each signature is assigned between two groups belonging to S0 and S3.

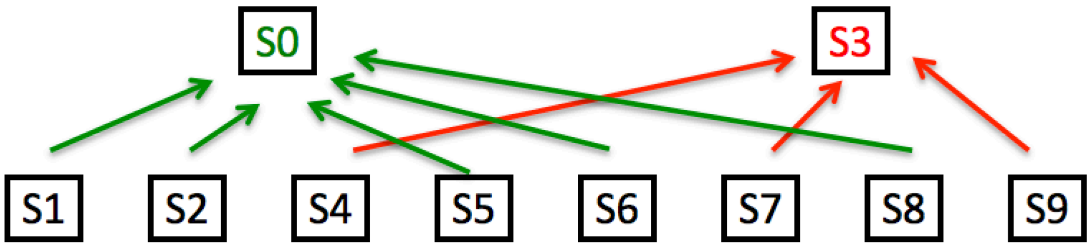


Figure 6.2 Distribution of Signatures into Two Groups Based on the Similarity at the Recursion Level 0.

We name the two groups – Group A and Group B. Once, the grouping is finished, each group size is checked with fmax. A group is finalized as a node if the size of that group is less than or equal to fmax, otherwise the algorithm is recursively called upon passing that group as the list of multi-set signatures.

In the below diagram, it is shown that both Group A and Group B have the sizes greater than fmax. There fore, the algorithm is called recursively upon both of them sequentially.

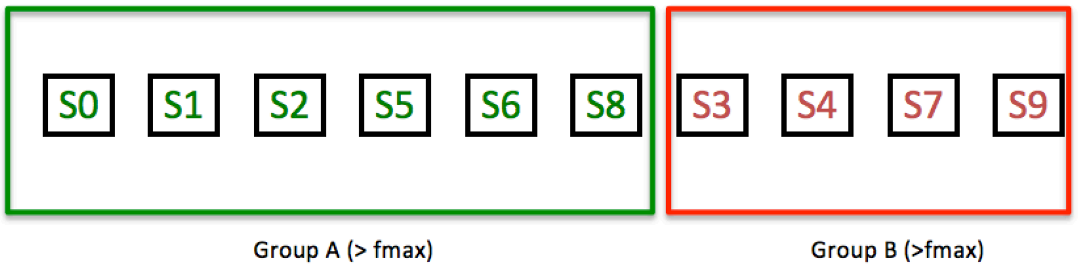


Figure 6.3. Fanout Size Checking for Each Group at Recursion Level 0.

6.4.2 Begin Processing Group A (Recursion level 1)

The algorithm is recursively called for Group A with the parameters A = {S0, S1, S2, S5, S6, S8}, fanout and k.

The K highest cardinality signatures are selected from S0, S1, S2, S5, S6 such that they are the most dissimilar signatures in the set.

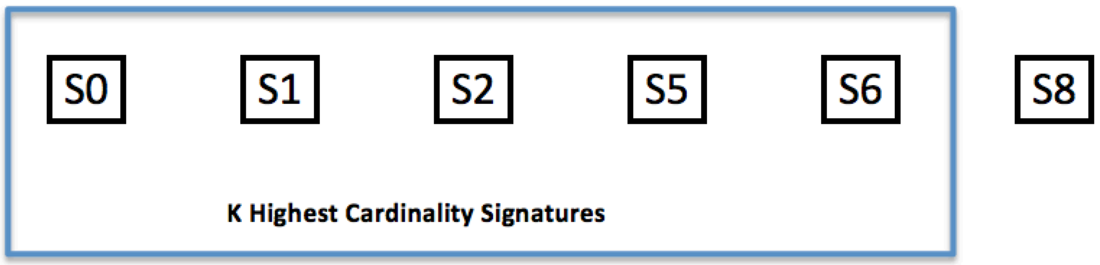


Figure 6.4. ‘K’ Highest Cardinality Signatures in Group A.

Let, S1 and S6 are chosen as the most dissimilar signatures from the above set. In this set the rest of the signatures in the list A are joined with one of the groups containing S1 or S6 based on their similarity.

Below diagram shows the iteration on list A where S0, S2, S5 and S8 are joined with the group they are most similar with.

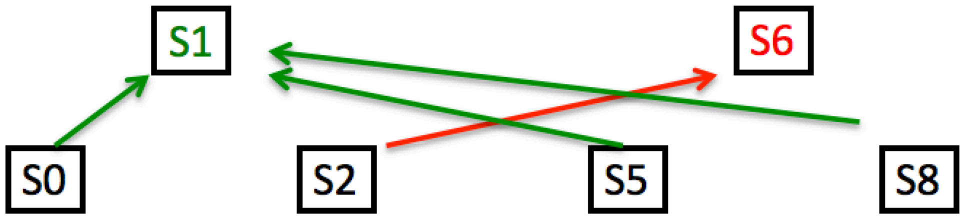


Figure 6.5 Distribution of Signatures into Two Groups Based on the Similarity at the Recursion Level 1.

Let the two groups be named A.A and A.B. In this step we compare the sizes of groups A.A and A.B to fmax value and recursively call the algorithm for the group which still has the size greater than fmax.

In the diagram below, Group A.A has higher size than fmax while group A.B is has size less than fmax. Therefore, the algorithm is recursively call on group A.A only.

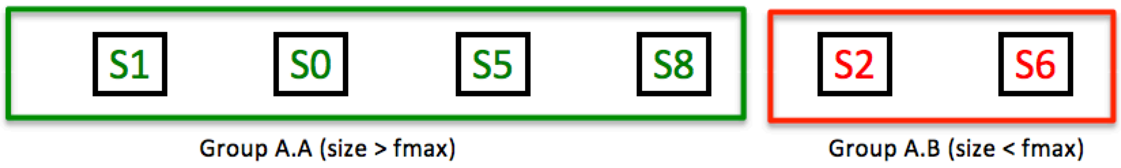


Figure 6.6. Fanout Size Checking for Each Group at Recursion Level 1.

6.4.3 Begin Processing Group A.A (Recursion level 2)

The algorithm is recursively called for Group A.A with the parameters A = {S1, S0, S5, S8}, fanout and k.

The K highest cardinality signatures are selected from S1, S0, S5, S8 such that they are the most dissimilar signatures in the set.

S1
S0
S5
S8
K Highest Cardinality Signatures

Figure 6.7. ‘K’ Highest Cardinality Signatures in Group A.A.

Let, S1 and S5 are chosen as the most dissimilar signatures from the above set. In this set the rest of the signatures in the list A are joined with one of the groups containing S1 or S5 based on their similarity.

Below diagram shows the iteration on list A where S0 and S8 are joined with the group they are most similar with.

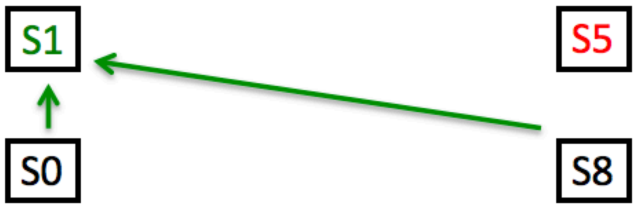


Figure 6.8. Distribution of Signatures into Two Groups Based on the Similarity at the Recursion Level 2.

Let the two groups be named A.A.A and A.B.B. In this step we compare the sizes of groups A.A.A and A.B.B to fmax value and recursively call the algorithm for the group which still has the size greater than fmax.

In the diagram below, both Group A.A.A and group A.B.B have the sizes less than fmax. Therefore, nodes are constructed on each of these groups after union operation applied on all the signatures within a group.

In the figure below it is shown that Node0 and Node1 are being constructed on group A.A.A and group A.B.B containing signatures S1, S0 & S8 and signature S5 respectively.

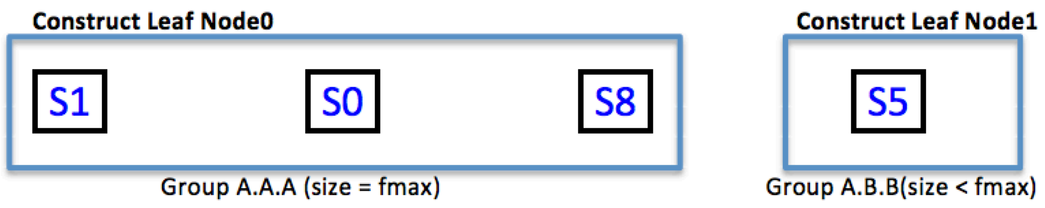


Figure 6.9. Fanout Size Checking for Each Group at Recursion Level 2.

6.4.4 End Processing Group A.A

Once the two nodes are formed, the algorithm for this recursion ends here.

6.4.5 Begin Processing Group A.B (Recursion level 2)

Once, the recursion level 2 algorithm gain controls after processing A.A, it creates a node for group A.B since its size is less than fmax.

Below diagram shows the construction of node2 on group A.B with the union of signatures S2 and S6.

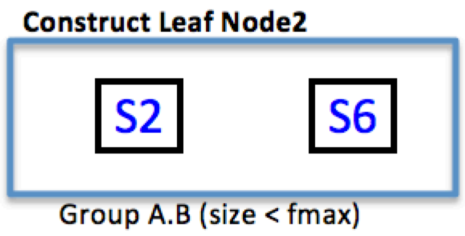


Figure 6.10. Constructing Leaf Node 2.

6.4.6 End Processing Group A.B

Once all the nodes are finished within the group A, the recursion level 1 gains control and begins to process group B.

6.4.7 Begin Processing Group B (Recursion level 1)

The algorithm is recursively called for Group B with the parameters A = {S3, S4, S7, S9}, fanout and k.

The K highest cardinality signatures are selected from S3, S4, S7, S9 such that they are the most dissimilar signatures in the set.

S3
S4
S7
S9
K Highest Cardinality Signatures

Figure 6.11. 'K' Highest Cardinality Signatures in Group B.

Let, S3 and S9 are chosen as the most dissimilar signatures from the above set. In this set the rest of the signatures in the list A are joined with one of the groups containing S3 or S9 based on their similarity.

Below diagram shows the iteration on list A where S4 and S7 are joined with the group they are most similar with.

Figure 6.12. Distribution of Signatures into Two Groups Based on the Similarity at the Recursion Level 1.

Let the two groups be named B.A and B.B. In this step we compare the sizes of groups B.A and B.B to fmax value.

In the diagram below, both Group B.A and group B.B have the sizes less than fmax. Therefore, nodes are constructed on each of these groups after union operation applied on all the signatures within a group.

In the figure below it is shown that Node3 and Node4 are being constructed on group B.A and group B.B containing signatures S3 & S4 and signature S7 & S9 respectively.

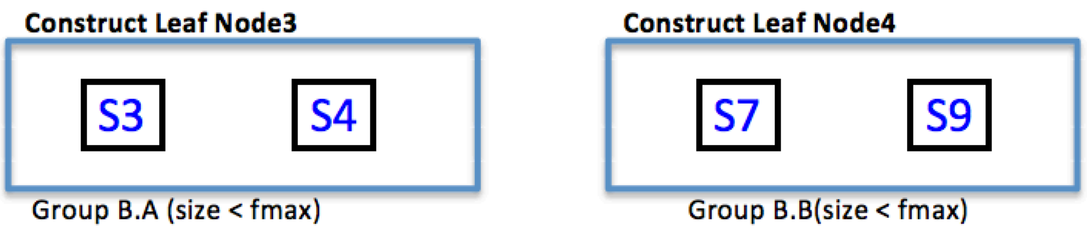


Figure 6.13. Fanout Size Checking for Each Group at Recursion Level 1.

6.4.8 End Processing Group B

Once all the nodes are finished within the group B, the recursion level 0 gains control.

6.4.9 End Processing all Groups

Here the control reaches to the base recursion level where it started. Since it completed processing both group A and Group B, the algorithm terminates successfully.

6.4.10 All leaf Nodes in the tree

Below is a figure showing all the leaf nodes for the example after the algorithm for leaf nodes generation finishes.

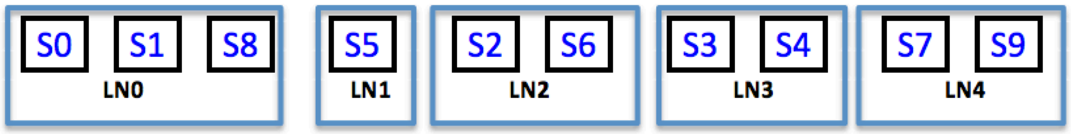


Figure 6.14. Showing all Finalized Leaf Nodes LN0, LN1, LN2, LN3, LN4.

Although ideally, we can construct more hierarchy in the tree for optimized performance, in our implementation we just construct the tree with one level only. Therefore, our tree has leaf nodes only.

6.5 Construction of Bloom Counters for Each Node in the tree

A bloom counter is constructed on each leaf node with the union of all signatures within. All the bloom counters have the same parameters and size. The parameters are chosen so that the performance is highest in terms of time taken to run a query. The number of false positives returned in the candidate matches should also be kept minimal. The figure below shows the bloom counters pointed by each leaf node in the tree.

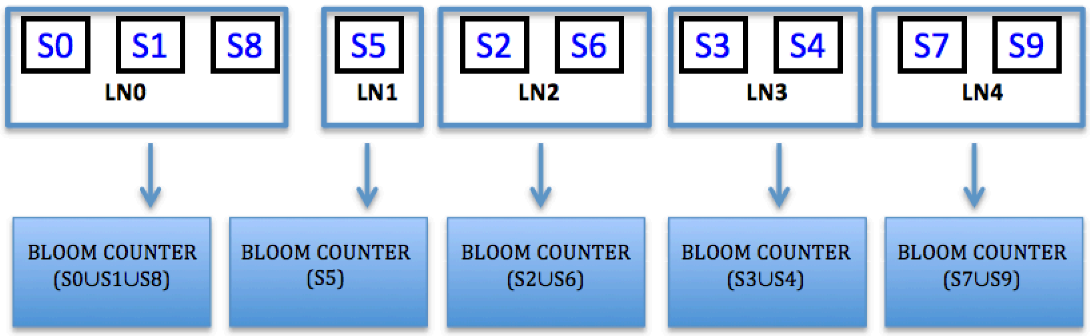


Figure 6.15. Construction of Bloom Counters Based on the Grouping of Signatures in Each Node.

CHAPTER 7

QUERYING THE SIGNATURE TREE

7.1 Our Querying Approach

Querying index with a given query is one of the most important processes involved in our system. One of the main goals our querying approach is to quickly identify and eliminate those parts of the dataset which we are sure will not return any results. This potentially can save a lot of time, as we do not have to run the query against entire dataset.

One-way to quickly eliminate those graphs which would not return any results is to run the query on only those graphs from huge dataset, which we know have the potential to return results. We call such graphs as the candidate matches.

There are a lot of parameters involved for optimizing the system in order to get the best candidate matches. One parameter is to have the number of candidate matches as low as possible. That means that we have successfully identified and eliminated most of the graphs from the huge collection graphs in the dataset. Another important parameter for effective querying is to have least number of false positives in the candidate matches. It is also possible that the entire candidate matches which were identified to return results may not be guarantee to return any results. There could be a few graphs which even though are identified as a candidate match could return no results at all when the query is run against that graph. We call such graphs as false positives. Lesser the number of false positives, more efficient is the querying approach.

7.2 Candidate Matches

Once the candidate matches are identified, the query is run against each of the graphs sequentially using a standard RDF data-querying tool. We then aggregate the results return by each graph and present it as the out put.

7.3 Query Classification

We classified RDF queries into two broad categories on the basis of their selectivity. Types of queries

- Low Selectivity
- High Selectivity

In general our approach has been specifically designed for queries with high selectivity rather than queries with low selectivity. We discuss about these types in detail in the next few sections with examples.

7.4 Low Selectivity Query

Low Selectivity Queries: Those queries, which return high number of results from a large dataset are classified as low selectivity queries. These queries tend to have more variables and less known attributes in them. These queries typically also have less number of joins which make them more generic and thus usually return many results.

Example:

SELECT ?student ?professor

WHERE {

?student <studies_at> "Unv" .

```
  ?professor <teaches> ?student .
}
```

As the above example shows the query returns all the pairs of students and their professors.

7.5 High Selectivity Queries

Those queries, which return low number of results from a large dataset, are classified as high selectivity queries. These queries tend to have lesser number of variables and larger number of uri's or known attributes in them. Also these queries have many number of joins which make the results much specific and refined.

Example:

```
SELECT ?student ?professor
WHERE {
  ?student <studies_at> "Unv" .
  ?student <enrolled> "course A" .
  ?student <batch_of> "2000" .
  ?student <teaching_assistant> "Math101" .
  ?student <author_of> "research1" .
  ?professor <teaches> ?student .
  ?professor <studied_at> "unv2" .
  ?professor <teaches> "Math101" .
  ?professor <research_interest> "Web Technologies" .
  ?professor <author_of> "research1" .
```

}

As shown in the above example, the query returns a very specific pairs of students and their professors who are met certain conditions like being the part of the same course, authors of same paper, having same research interest etc.

7.8 High Selectivity Querying Challenges

In the real time scenario, high selectivity queries are more expected as they are more useful while querying a large dataset for a specific information. At the same time, it is more challenging to return results for the query with very high selectivity compared to its counter part. One of the main reasons why running a query with high selectivity is challenging because of the number of joins it contains. Most of the existing query systems use keyword matches to find results and more joins mean more keyword searches. Hence, systems performing big queries take a huge amount of time to return the results compared to smaller ones. Sometimes, the queries can get highly selective and the number of joins can get as high as fifteen to twenty.

Our system has been designed to address specifically this problem of querying large datasets with queries containing many number of joins. One of the important aspect of our system design that helps us overcome this problem is we perform graph based searching rather than purely rely on keyword based searching. In addition to this, our smart graph preprocessing and indexing system gives us a huge advantage with queries which have large number of joins.

We have tested our system a wide range of queries. A few queries contain as many as twenty joins where as few queries contain a about just five joins. We use our system a platform on which a couple of other tools run. We use Jira on the top of our system for queries with high selectivity while we use RDF-3X on the top of our system when the queries have low selectivity. We have benchmarked the whole system against a tool called RDF-3X which is considered as a standard to query RDF data.

7.9 Sample Queries

Q1 (BIG) # *Querying for a professor who is an advisor to the given set of students*

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?professor ?course ?email ?phone ?research ?udergradUnv ?masterUnv
?PhdUnv WHERE
{
        # Professor details
        ?professor  rdf:type ub:FullProfessor .
        ?professor ub:name "FullProfessor1" .
        ?professor ub:teacherOf ?course .
        ?professor ub:undergraduateDegreeFrom ?undergradUnv  .
        ?professor ub:mastersDegreeFrom ?masterUnv .
        ?professor ub:doctoralDegreeFrom ?PhdUnv .
        ?professor ub:worksFor <http://www.Department12.University1.edu> .
        ?professor ub:emailAddress ?email .
        ?professor ub:telephone ?phone .
        ?professor ub:researchInterest ?research .
          <http://www.Department12.University1.edu/FullProfessor1/Publication2>
ub:publicationAuthor ?professor .
        <http://www.Department12.University1.edu/FullProfessor1/Publication17>
ub:publicationAuthor ?professor .
```

```
        # List of students
        <http://www.Department12.University1.edu/UndergraduateStudent36>
ub:advisor ?professor .
        <http://www.Department12.University1.edu/UndergraduateStudent170>
ub:advisor ?professor .
        <http://www.Department12.University1.edu/GraduateStudent41> ub:advisor
?professor .
        <http://www.Department12.University1.edu/GraduateStudent65> ub:advisor
?professor .

}
```

Q2 (BIG) # Querying for the co-authors of research papers such that there is a student-professor relationship between the co-authors.

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x ?name1 ?unv ?Email1 ?phone1 ?course  ?professor ?name2 ?unv2 ?unv3
?Email2 ?phon2 ?RearchInt ?course2 ?publication1  WHERE
{
        # Student details
        ?x rdf:type ub:GraduateStudent> .
        ?x ub:name ?name1 .
        ?x ub:memberOf <http://www.Department0.University1167.edu> .
        ?x ub:undergraduateDegreeFrom ?univ .
        ?x ub:emailAddress ?Email1 .
        ?x ub:telephone ?phone1 .
        ?x ub:takesCourse ?course .
        ?x ub:advisor ?professor .
        ?x rdf:type ub:ResearchAssistant .
        ?publication1 ub:publicationAuthor ?x .
        # Professor Details
        ?professor rdf:type ub:AssociateProfessor .
        ?professor ub:name ?name2 .
        ?professor ub:undergraduateDegreeFrom ?univ1 .
        ?professor ub:mastersDegreeFrom ?univ2 .
        ?professor ub:doctoralDegreeFrom ?univ3 .
        ?professor ub:worksFor <http://www.Department0.University1167.edu> .
        ?professor ub:emailAddress ?Email2 .
        ?professor ub:telephone ?phone2 .
        ?professor ub:researchInterest ?ResearchInt .
        ?professor ub:teacherOf ?course2 .
        ?publication1 ub:publicationAuthor ?professor .
}
```

Q3 (BIG) # Querying for two colleagues (professors) who also studied in the same college.

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?professor1 ?unv1 ?unv5 ?univ2 ?name1 ?Email1 ?phon1 ?ResearchInt1
?course1 ?publication1 ?professor2 ?univ4 ?name2 ?Email2 ?phone2 ?ResearchInt2
WHERE
{

        # Professor_1 Details
        ?professor1 rdf:type  ub:FullProfessor> .
        ?professor1 ub:undergraduateDegreeFrom ?univ1 .
        ?professor1 ub:mastersDegreeFrom ?univ1 .
        ?professor1 ub:doctoralDegreeFrom ?univ5 .
        ?professor1 ub:worksFor ?univ2 .
        ?professor1 ub:name ?name1 .
        ?professor1 ub:emailAddress ?Email1 .
        ?professor1 ub:telephone ?phone1 .
        ?professor1 ub:researchInterest ?ResearchInt1 .
        ?professor1 ub:teacherOf ?course1 .
        ?publication1 ub:publicationAuthor ?professor1 .

        # Professor_2 Details
        ?professor2 rdf:type  ub:AssociateProfessor .
        ?professor2 ub:undergraduateDegreeFrom ?univ1 .
        ?professor2 ub:mastersDegreeFrom ?univ1 .
        ?professor2 ub:doctoralDegreeFrom ?univ4 .
        ?professor2 ub:worksFor ?univ2 .
        ?professor2 ub:name ?name2 .
        ?professor2 ub:emailAddress ?Email2 .
        ?professor2 ub:telephone ?phone2 .
        ?professor2 ub:researchInterest ?ResearchInt2 .
        ?professor2 ub:teacherOf ?course2 .
        ?publication2 ub:publicationAuthor ?professor2 .
}
```

<u>Q5 (SMALL)</u> # Querying for a student.

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x ?dep ?univ WHERE
{
        ?x rdf:type ub:GraduateStudent .
        ?x ub:undergraduateDegreeFrom ?univ .
        ?univ rdf:type ub:University .
        ?dep rdf:type ub:Department .
        ?x ub:memberOf ?dep .
        ?dep ub:subOrganizationOf ?univ .
}
```

CHAPTER 8

EVALUATION

8.1 Implementation

In this section we discuss our implementation and evaluation of our tool against one of the best RDF-3x tools in the recent times. Our entire system was built in C++ using g++ 4.6.1 on Ubuntu 11.10. We used a local box with a four core 3.30GHz each processor and 8GB primary memory.

We performed our experiments on a synthetic dataset called LUBM. The dataset contains a set of small graphs which we call as the connected components. The entire dataset contains about 50 Billion triples in it.

8.2 Supplementary Tools Used

For querying we used JenaTDB on the top of our tool for queries with large selectivity and RDF-3x on the top of our tool for queries with low selectivity. We compared the results of the performance of each query with our approach verses the RDF-3x tool run as a stand alone tool.

- Tools used
    - JenaTDB
    - RDF-3X

CHAPTER 9

EXPERIMENTS

Thorough experiments were performed to compare our approach with the stand alone RDF-3X tool which is considered to be a benchmark among all the contemporary tools to query RDF data. The experiments were focused in varying parameters like fanout size, seed pick size.

This section is divided into two subsections. In the first subsection we discuss how our tool performed with varying Seed Pick Size. In the second subsection we discuss our performance with varying Seed Pick Size and in the last subsection we vary Size per Bloom Counter. In each of these subsections we draw a conclusion to determine the value of that subsection's parameter which boosts our performance the best. We carry on the best value for a particular subsection to the subsequent subsections.

A quick note to understand is in each of our experiments we always put special emphasis on how we perform against big queries than small queries as our whole approach was developed for those queries which take huge processing times using conventional approaches.

9.1 Experiment Set 1 – Varying Fanout Size

In this subsection we vary the Fanout Size in 250, 500, 1000 & 2000 while keeping the other parameters constant. At the end of this experiment set, we will conclude by picking the Fanout Size that gives us maximum performance boost.

9.1.1 Fanout Size : 250 (LUBM1)

This is the experiment where we assign the Fanout Size a value of 250. We now discuss how it performed while building the index, querying the index with three big queries and one small query and finally we compare our performance to that of RDF-3X when used as a stand alone tool.

Below are the timings found for each step in our index building process. The disk spaces for storing each index are also shown. We performed this experiment with 50 Billion RDF triples taken as the input.

Table 9.1.1 Indexing Evaluation for Fanout as 250.

Description:	**Indexing Evaluation for Fanout : 250**						
Fanout Size	**Buffers #**	**Seed Pick Size**	**Bloom Counters #**	**Bloom Functions #**	**Each Bloom Counter Size**	**Input RDF Partitions Disk Size**	**Input RDF Partitions Triples #**
250	32768	200	90 Million	4	8	47 GB	50 Billion
RDF Format to Signatures Conversion Time	**Bloom Counters Construction Time**	**Signature Groups to RDF Format Conversion Time**	**Jena TDB Construction Time**	**RDF-3X Construction Time**	**Total Index Construction Time**		
12764.69	7629.13	1535	9020.38	2182	33131.2		
# of Blooms Constructed	**Bloom Counters Disk Size**	**Jena TDB Index Disk Size**	**RDF-3X Index Disk Size**				
322	27 GB	25 GB	14 GB				

Below are the querying timings. The timings for Candidates Identifying and Index Querying for those candidates are shown separately. We also used different approaches by classifying the query types as Big or Small based on the Number of Triples they contain.

Table 9.1.2 Querying Evaluation for Fanout as 250.

Description:		**Querying Evaluation for Fanout : 250**						
Query	**# of Triples in the Query**	**Query Type**	**Approach Used**	**# of Candidate Matches**	**# of False Positives**	**Candidates Identifying Time**	**Index Querying Time**	**Total Querying Time (Identifying + Querying)**
Q1	16	Big	RIS + Jena TDB	1	0	0.082	3.373	3.455
Q2	21	Big	RIS + Jena TDB	1	0	0.080	3.353	3.433
Q3	22	Big	RIS + Jena TDB	2	2	0.057	5.386	5.443
Q4	5	Small	RIS + **RDF-3X**	322	15	0.105	111.538	111.643

Below is a detailed comparison between the performance of our tool to that of RDF-3X as stand alone for each query. We can observe for all the queries including both big and small type, our approach out performs RDF-3X stand alone approach. Our approach is significantly efficient for bigger queries specifically.

Table 9.1.3 Performance Comparison for Fanout as 250.

Description:	**Performance Comparison for Fanout : 250**			
Query	**# of Results**	**Total Querying Time (Identifying + Querying)**	**RDF-3x Only Time**	**Performance Improvement**
Q1	4	3.455	22.261	644%
Q2	10	3.433	3000.922	87413%
Q3	0	5.443	10839.45	199144%
Q4	2481	111.643	216.371	193%

9.1.2 Fanout Size : 500 (LUBM2)

Now we change the Fanout Size value to 500 and repeat the same experiment. Below are the timings found for each step in our index building process. The disk spaces for each index are also shown.

Table 9.1.4 Indexing Evaluation for Fanout as 500.

Description:	**Indexing Evaluation for Fanout : 500**						
Fanout Size	**Buffers #**	**Seed Pick Size**	**Bloom Counters #**	**Bloom Functions #**	**Each Bloom Counter Size**	**Input RDF Partitions Disk Size**	**Input RDF Partitions Triples #**
500	32768	200	90 Million	4	8	47 GB	50 Billion

Description:	**Indexing Evaluation for Fanout : 500**						
RDF Format to Signatures Conversion Time	**Bloom Counters Construction Time**	**Signature Groups to RDF Format Conversion Time**	**Jena TDB Construction Time**	**RDF-3x Construction Time**	**Total Index Construction Time**		
12764.69	16285.3	1759	8362.15	2525	41696.14		
# of Blooms Constructed	**Bloom Counters Disk Size**	**Jena TDB Index Disk Size**	**RDF-3X Index Disk Size**				
180	16 GB	25 GB	14 GB				

Below are the querying timings. The timings for Candidates Identifying and Index Querying for those candidates are shown separately. We also used different approaches by classifying the query types as Big or Small based on the Number of Triples they contain.

For all the querying experiments, the queries were run three times on the index while clearing the cache after each run. Thus the readings are a result of an average of three readings. Also, it has to be noted that there were no major disparities between the three readings.

Table 9.1.5 Querying Evaluation for Fanout as 500.

Description:	Querying Evaluation for Fanout : 500							
Query	**# of Triples in the Query**	**Query Type**	**Approach Used**	**# of Candidate Matches**	**# of False Positives**	**Candidates Identifying Time**	**Index Querying Time**	**Total Querying Time (Identifying + Querying)**
Q1	16	Big	RIS + Jena TDB	1	0	0.082	3.253	3.335
Q2	21	Big	RIS + Jena TDB	1	0	0.217	4.052	4.269
Q3	22	Big	RIS + Jena TDB	4	4	0.052	12.653	12.705
Q4	5	Small	RIS + RDF-3X	179	10	0.033	74.864	74.897

Below is a detailed comparison between the performance of our tool to that of RDF-3X as stand alone for each query. We can observe for all the queries including both big and small type, our approach out performs RDF-3X stand alone approach. Our approach is significantly efficient for bigger queries specifically, since we do not rely on relational databases for generating a query plan. Big queries are expected to have inner many joins in them.

Table 9.1.6 Performance Comparison for Fanout as 500.

Description:	Performance Comparison for Fanout : 500			
Query	# of Results	Total Querying Time (Identifying + Querying)	RDF-3x Only Time	Performance Improvement
Q1	4	3.335	22.261	667%
Q2	10	4.269	3000.922	70295%
Q3	0	12.705	10839.45	85316%
Q4	2481	74.897	216.371	288%

9.1.3 Fanout Size : 1000 (LUBM3)

Now we change the Fanout Size value to 1000 and repeat the same experiment. Below are the timings found for each step in our index building process. The disk spaces for each index are also shown.

Table 9.1.7 Indexing Evaluation for Fanout as 1000.

Description:	Indexing Evaluation for Fanout : 1000						
Fanout Size	Buffers #	Seed Pick Size	Bloom Counters #	Bloom Functions #	Each Bloom Counter Size	Input RDF Partitions Disk Size	Input RDF Partitions Triples #
1000	32768	200	90 Million	4	8	47 GB	50 Billion

Description:	Indexing Evaluation for Fanout : 1000						
RDF Format to Signatures Conversion Time	Bloom Counters Construction Time	Signature Groups to RDF Format Conversion Time	Jena TDB Construction Time	RDF-3X Construction Time	Total Index Construction Time		
12764.69	32961.5	1586	8326.64	2375	58013.83		
# of Blooms Constructed	Bloom Counters Disk Size	Jena TDB Index Disk Size	RDF-3X Index Disk Size				
101	8.5 GB	25 GB	14 GB				

Below are the querying timings. The timings for Candidates Identifying and Index Querying for those candidates are shown separately. We also used different approaches by classifying the query types as Big or Small based on the Number of Triples they contain.

Table 9.1.8 Querying Evaluation for Fanout as 1000.

Description:		Querying Evaluation for Fanout : 1000						
Query	# of Triples in the Query	Query Type	Approach Used	# of Candidate Matches	# of False Positives	Candidates Identifying Time	Index Querying Time	Total Querying Time (Identifying + Querying)

Description:	Querying Evaluation for Fanout : 1000							
Q1	16	Big	RIS + Jena TDB	1	0	0.004	3.321	3.325
Q2	21	Big	RIS + Jena TDB	1	0	0.005	3.763	3.768
Q3	22	Big	RIS + Jena TDB	3	3	0.006	15.994	16.000
Q4	5	Small	RIS + RDF-3X	101	3	0.026	54.77	54.796

Below is a detailed comparison between the performance of our tool to that of RDF-3X as stand alone for each query. We can observe for all the queries including both big and small type, our approach out performs RDF-3X stand alone approach. Our approach is significantly efficient for bigger queries specifically.

Table 9.1.9 Performance Comparison for Fanout as 1000.

Description:	**Performance Comparison for Fanout : 1000**			
Query	**# of Results**	**Total Querying Time (Identifying + Querying)**	**RDF-3x Only Time**	**Performance Improvement**
Q1	4	3.325	22.261	669%
Q2	10	3.768	3000.922	79642%
Q3	0	16.000	10839.45	67746%

Description:	Performance Comparison for Fanout : 1000			
Q4	2481	54.796	216.371	394%

9.1.4 Fanout Size : 2000 (LUBM4)

Now we change the Fanout Size value to 2000 and repeat the same experiment. Below are the timings found for each step in our index building process. The disk spaces for each index are also shown.

Table 9.1.10 Indexing Evaluation for Fanout as 2000.

Description:	Indexing Evaluation for Fanout : 2000						
Fanout Size	**Buffers #**	**Seed Pick Size**	**Bloom Counters #**	**Bloom Functions #**	**Each Bloom Counter Size**	**Input RDF Partitions Disk Size**	**Input RDF Partitions Triples #**
2000	32768	200	90 Million	4	8	47 GB	50 Billion
RDF Format to Signatures Conversion Time	**Bloom Counters Construction Time**	**Signature Groups to RDF Format Conversion Time**	**Jena TDB Construction Time**	**RDF-3X Construction Time**	**Total Index Construction Time**		
12764.69	69883	1691	8361.44	2578	95278.13		
# of Blooms Constructed	**Bloom Counters Disk Size**	**Jena TDB Index Disk Size**	**RDF-3X Index Disk Size**				

Description:	Indexing Evaluation for Fanout : 2000						
59	5 GB	25 GB	14 GB				

Below are the querying timings. The timings for Candidates Identifying and Index Querying for those candidates are shown separately. We also used different approaches by classifying the query types as Big or Small based on the Number of Triples they contain.

For all the querying experiments, the queries were run three times on the index while clearing the cache after each run. Thus the readings are a result of an average of three readings. Also, it has to be noted that there were no major disparities between the three readings.

Table 9.1.11 Querying Evaluation for Fanout as 2000.

Description:		Querying Evaluation for Fanout : 2000						
Query	# of Triples in the Query	Query Type	Approach Used	# of Candidate Matches	# of False Positives	Candidates Identifying Time	Index Querying Time	Total Querying Time (Identifying + Querying)
Q1	16	Big	RIS + Jena TDB	1	0	0.007	3.248	3.255

Description:	Querying Evaluation for Fanout : 2000							
Q2	21	Big	RIS + Jena TDB	4	3	0.038	5.207	5.245
Q3	22	Big	RIS + Jena TDB	7	7	0.004	71.698	71.702
Q4	5	Small	RIS + RDF-3X	59	2	0.003	47.234	47.237

Below is a detailed comparison between the performance of our tool to that of RDF-3X as stand alone for each query. We can observe for all the queries including both big and small type, our approach out performs RDF-3X stand alone approach. Our approach is significantly efficient for bigger queries specifically.

Table 9.1.12 Performance Comparison for Fanout as 2000.

Description:	**Performance Comparison for Fanout : 2000**			
Query	**# of Results**	**Total Querying Time (Identifying + Querying)**	**RDF-3x Only Time**	**Performance Improvement**
Q1	4	3.255	22.261	683%
Q2	10	5.245	3000.922	57214%
Q3	0	71.702	10839.45	15117%
Q4	2481	47.237	216.371	458%

9.1.5 Conclusion for Fanout Size

The indexing times have increased as the fanout size increased. Understandably, as the fanout value increased, the number of bloom counters to be constructed decreased.

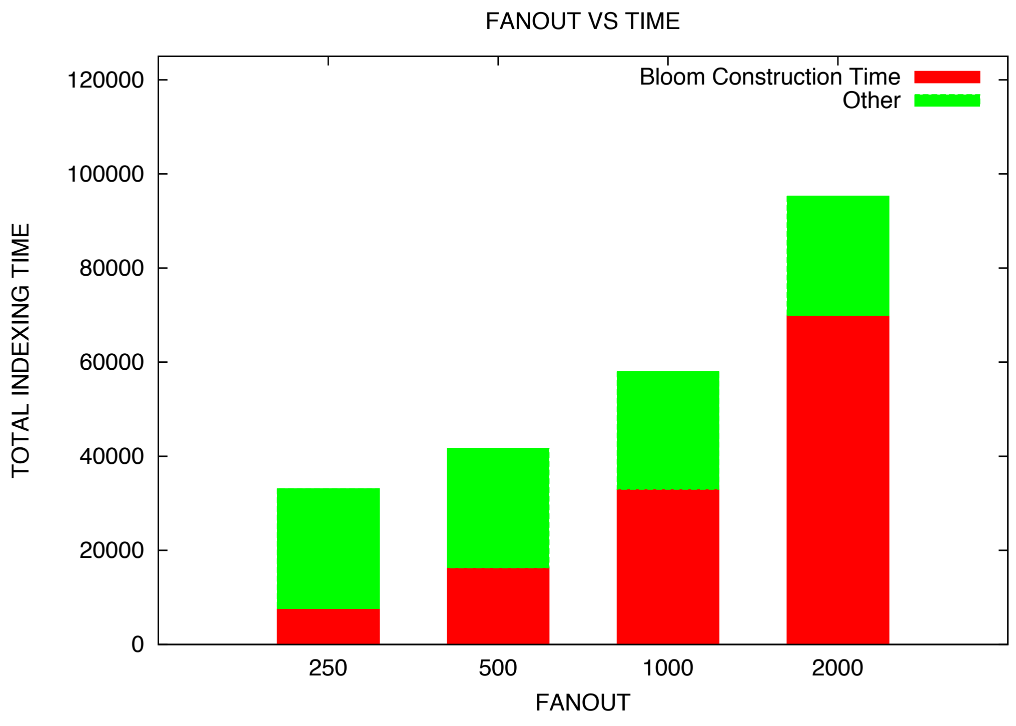


Figure 8.1 A graph between Fanout and Total Indexing Time.

It can be observed from the above graph that increase in the time period is attributed mostly to the bloom counters construction time while the other index construction time remained almost the same irrespective of the Fanout value.

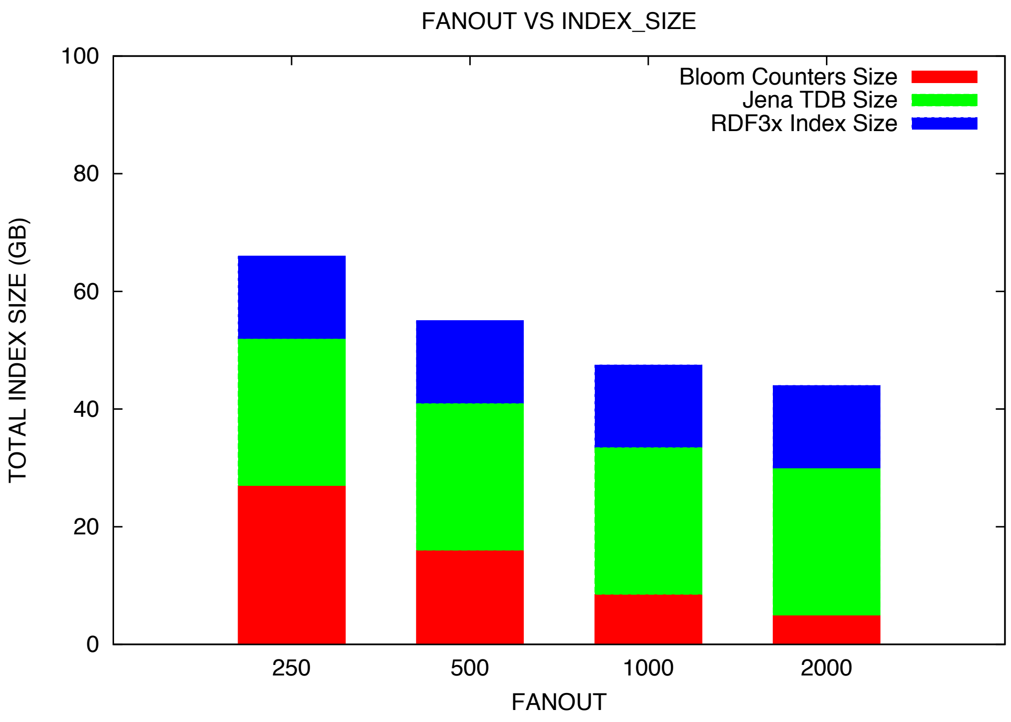


Figure 8.2 A graph between Fanout and Total Indexing Time.

From the above four experiments in this subsection, it can be inferred that the querying times for large queries (Q1, Q2 & Q3) remained small when the fanout was small where as the small query (Q4) performed much better when the fanout was the maximum.

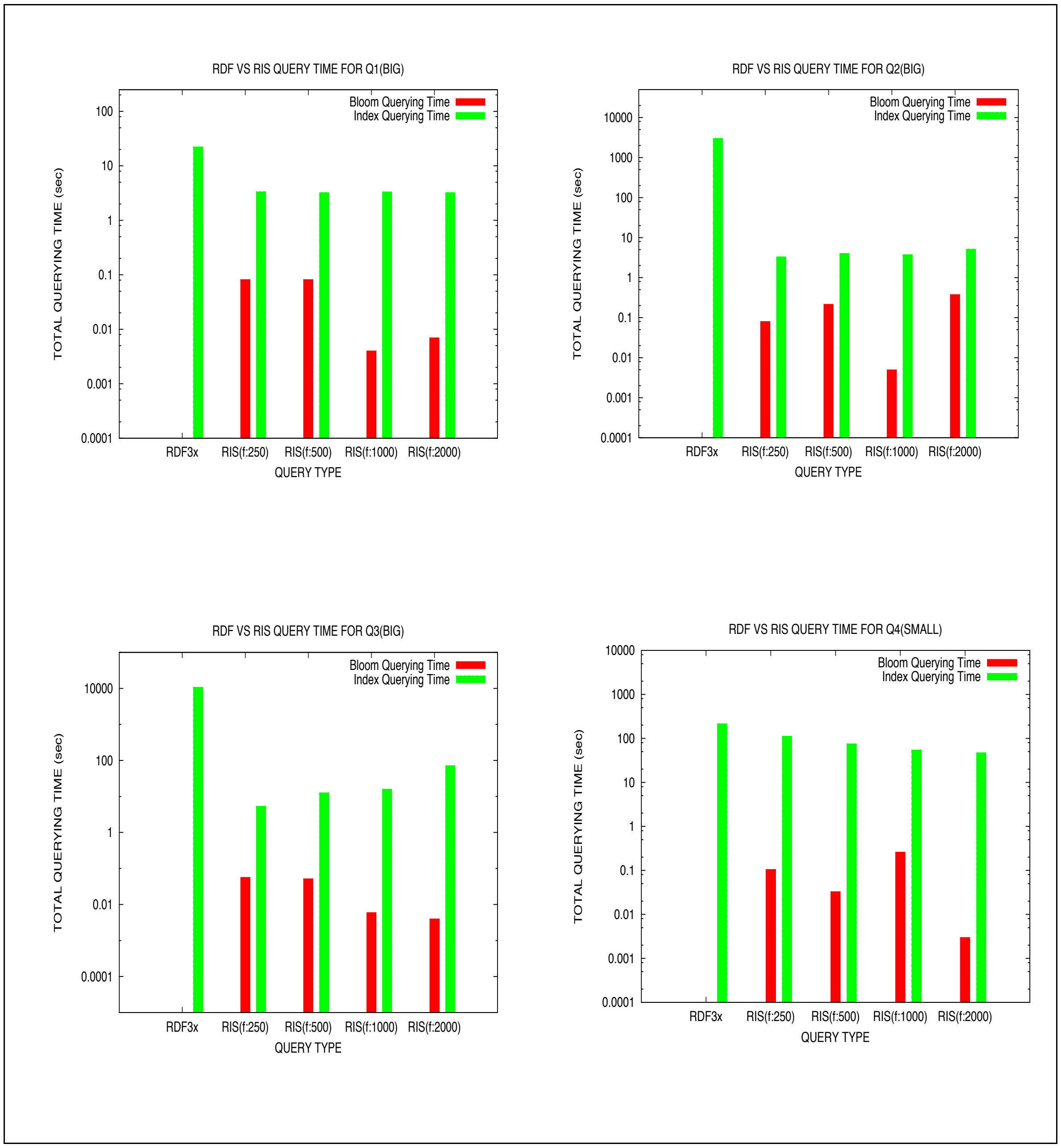


Figure 8.3 Graphs between the Query Type for different Fanout values and Total Querying Time for queries Q1, Q2, Q3 and Q4.

In conclusion, the performance of our approach for the big queries is best when the fanout was least i.e. 250. Therefore, we fix fanout value to be 250 for our remaining experiments.

9.2 Experiment Set 2 – Varying Seed Pick Size

As we determined the optimal fanout size to be 250, we will now perform a series of experiments to determine the optimal Seed Pick Size. In this subsection we vary the Seed Pick Size in 100, 200, 400 & 800 while keeping the other parameters constant. At the end of this experiment set, we will conclude by picking the Seed Pick Size that gives us maximum performance boost.

9.2.1 Seed Pick Size : 100 (LUBM7)

In this experiment we assign the Seed Pick Size a value of 100. We will now discuss how it performed while building the index, querying the index with three big queries and one small query and finally we compare our performance to that of RDF-3X when used as a stand alone tool.

Below are the timings found for each step in our index building process. The disk spaces for storing each index are also shown. We performed this experiment with 50 Billion RDF triples taken as the input.

Table 9.2.1 Indexing Evaluation for Seed Pick Size as 100.

Description:	**Indexing Evaluation for Seed Pick Size : 100**						
Fanout Size	**Buffers #**	**Seed Pick Size**	**Bloom Counters #**	**Bloom Functions #**	**Each Bloom Counter Size**	**Input RDF Partitions Disk Size**	**Input RDF Partitions Triples #**
250	32768	100	90 Million	4	8	47 GB	50 Billion

Description:	**Indexing Evaluation for Seed Pick Size : 100**						
RDF Format to Signatures Conversion Time	**Bloom Counters Construction Time**	**Signature Groups to RDF Format Conversion Time**	**Jena TDB Construction Time**	**RDF-3X Construction Time**	**Total Index Construction Time**		
12764.69	7744.11	1753	9075.17	2359	33695.97		
# of Blooms Constructed	**Bloom Counters Disk Size**	**Jena TDB Index Disk Size**	**RDF-3X Index Disk Size**				
334	28 GB	25 GB	14 GB				

Below are the querying timings. The timings for Candidates Identifying and Index Querying for those candidates are shown separately. We also used different approaches by classifying the query types as Big or Small based on the Number of Triples they contain.

For all the querying experiments, the queries were run three times on the index while clearing the cache after each run. Thus the readings are a result of an average of three readings. Also, it has to be noted that there were no major disparities between the three readings.

Table 9.2.2 Querying Evaluation for Seed Pick Size as 100.

Description:		Querying Evaluation for Seed Pick Size : 100						
Query	**# of Triples in the Query**	**Query Type**	**Approach Used**	**# of Candidate Matches**	**# of False Positives**	**Candidates Identifying Time**	**Index Querying Time**	**Total Querying Time (Identifying + Querying)**
Q1	16	Big	RIS + Jena TDB	1	0	0.060	4.604	4.664
Q2	21	Big	RIS + Jena TDB	1	0	0.070	4.396	4.466
Q3	22	Big	RIS + Jena TDB	2	2	0.116	5.049	5.165
Q4	5	Small	RIS + RDF-3X	334	35	0.152	111.538	111.690

Below is a detailed comparison between the performance of our tool to that of RDF-3X as stand alone for each query. We can observe for all the queries including both big and small type, our approach out performs RDF-3X stand alone approach. Our approach is significantly efficient for bigger queries specifically.

For all the querying experiments, the queries were run three times on the index while clearing the cache after each run. Thus the readings are a result of an average of three readings. Also, it has to be noted that there were no major disparities between the three readings.

Table 9.2.3 Performance Comparison for Seed Pick Size as 100.

Description:	Performance Comparison for Seed Pick Size : 100			
Query	**# of Results**	**Total Querying Time (Identifying + Querying)**	**RDF-3x Only Time**	**Performance Improvement**
Q1	4	4.664	22.261	477%
Q2	10	4.466	3000.922	67194%
Q3	0	5.165	10839.45	209863%
Q4	2481	111.690	216.371	193%

9.2.2 Seed Pick Size : 200 (LUBM1)

In this experiment we assign the Seed Pick Size a value of 200. We will now discuss how it performed while building the index, querying the index with three big queries and one small query and finally we compare our performance to that of RDF-3X when used as a stand alone tool.

Table 9.2.4 Indexing Evaluation for Seed Pick Size as 200.

Description:	Indexing Evaluation for Seed Pick Size : 200						
Fanout Size	**Buffers #**	**Seed Pick Size**	**Bloom Counters #**	**Bloom Functions #**	**Each Bloom Counter Size**	**Input RDF Partitions Disk Size**	**Input RDF Partitions Triples #**
250	32768	200	90 Million	4	8	47 GB	50 Billion

Description:	Indexing Evaluation for Seed Pick Size : 200						
RDF Format to Signatures Conversion Time	Bloom Counters Construction Time	Signature Groups to RDF Format Conversion Time	Jena TDB Construction Time	RDF-3X Construction Time	Total Index Construction Time		
12764.69	7629.13	1535	9020.38	2182	33131.2		
# of Blooms Constructed	Bloom Counters Disk Size	Jena TDB Index Disk Size	RDF-3X Index Disk Size				
322	27 GB	25 GB	14 GB				

Below are the querying timings. The timings for Candidates Identifying and Index Querying for those candidates are shown separately.

Table 9.2.5 Querying Evaluation for Seed Pick Size as 200.

Description:		Querying Evaluation for Seed Pick Size : 200						
Query	# of Triples in the Query	Query Type	Approach Used	# of Candidate Matches	# of False Positives	Candidates Identifying Time	Index Querying Time	Total Querying Time (Identifying + Querying)
Q1	16	Big	RIS + Jena TDB	1	0	0.082	3.373	3.455
Q2	21	Big	RIS + Jena TDB	1	0	0.080	3.353	3.433

Description:	**Querying Evaluation for Seed Pick Size : 200**							
Q3	22	Big	RIS + Jena TDB	2	2	0.057	5.386	5.443
Q4	5	Small	RIS + RDF-3X	322	15	0.105	111.538	111.643

Below is a detailed comparison between the performance of our tool to that of RDF-3X as stand alone for each query. We can observe for all the queries including both big and small type, our approach out performs RDF-3X stand alone approach. Our approach is significantly efficient for bigger queries specifically.

Table 9.2.6 Performance Comparison for Seed Pick Size as 200.

Description:	**Performance Comparison for Seed Pick Size : 200**			
Query	**# of Results**	**Total Querying Time (Identifying + Querying)**	**RDF-3x Only Time**	**Performance Improvement**
Q1	4	3.455	22.261	644%
Q2	10	3.433	3000.922	87413%
Q3	0	5.443	10839.45	199144%
Q4	2481	111.643	216.371	193%

9.2.3 Seed Pick Size : 400 (LUBM6)

In this experiment we assign the Seed Pick Size a value of 400. We will now discuss how it performed while building the index, querying the index with three big queries and one small query and finally we compare our performance to that of RDF-3X when used as a stand alone tool.

Below are the timings found for each step in our index building process. The disk spaces for storing each index are also shown. We performed this experiment with 50 Billion RDF triples taken as the input.

Table 9.2.7 Indexing Evaluation for Seed Pick Size as 400.

Description:	**Indexing Evaluation for Seed Pick Size : 400**						
Fanout Size	**Buffers #**	**Seed Pick Size**	**Bloom Counters #**	**Bloom Functions #**	**Each Bloom Counter Size**	**Input RDF Partitions Disk Size**	**Input RDF Partitions Triples #**
250	32768	400	90 Million	4	8	47 GB	50 Billion
RDF Format to Signatures Conversion Time	**Bloom Counters Construction Time**	**Signature Groups to RDF Format Conversion Time**	**Jena TDB Construction Time**	**RDF-3X Construction Time**	**Total Index Construction Time**		
12764.69	7652.56	1725	9006.87	2362	33511.12		
# of Blooms Constructed	**Bloom Counters Disk Size**	**Jena TDB Index Disk Size**	**RDF-3X Index Disk Size**				
315	27 GB	25 GB	14 GB				

Below are the querying timings. The timings for Candidates Identifying and Index Querying for those candidates are shown separately. We also used different approaches by classifying the query types as Big or Small based on the Number of Triples they contain.

Table 9.2.8 Querying Evaluation for Seed Pick Size as 400.

Description:		**Querying Evaluation for Seed Pick Size : 400**						
Query	**# of Triples in the Query**	**Query Type**	**Approach Used**	**# of Candidate Matches**	**# of False Positives**	**Candidates Identifying Time**	**Index Querying Time**	**Total Querying Time (Identifying + Querying)**
Q1	16	Big	RIS + Jena TDB	1	0	0.093	3.433	3.526
Q2	21	Big	RIS + Jena TDB	1	0	0.047	3.229	3.276
Q3	22	Big	RIS + Jena TDB	2	2	0.07	4.561	4.631
Q4	5	Small	RIS + RDF-3X	59	23	0.093	119.198	119.291

Below is a detailed comparison between the performance of our tool to that of RDF-3X as stand alone for each query. We can observe for all the queries including both big and small type, our approach out performs RDF-3X stand alone approach. Our approach is significantly efficient for bigger queries specifically.

Table 9.2.9 Performance Comparison for Seed Pick Size as 400.

Description:	Performance Comparison for Seed Pick Size : 400			
Query	**# of Results**	**Total Querying Time (Identifying + Querying)**	**RDF-3x Only Time**	**Performance Improvement**
Q1	4	3.526	22.261	631%
Q2	10	3.276	3000.922	91603%
Q3	0	4.631	10839.45	234062%
Q4	2481	119.291	216.371	181%

9.2.4 Seed Pick Size : 800 (LUBM5)

In this experiment we assign the Seed Pick Size a value of 800. We will now discuss how it performed while building the index, querying the index with three big queries and one small query and finally we compare our performance to that of RDF-3X when used as a stand alone tool.

Below are the timings found for each step in our index building process. The disk spaces for storing each index are also shown. We performed this experiment with 50 Billion RDF triples taken as the input. All the timings are shown in seconds by default and are an average of three readings taken by clearing out the system cache before each experiment.

Table 9.2.10 Indexing Evaluation for Seed Pick Size as 800.

Description:	**Indexing Evaluation for Seed Pick Size : 800**						
Fanout Size	**Buffers #**	**Seed Pick Size**	**Bloom Counters #**	**Bloom Functions #**	**Each Bloom Counter Size**	**Input RDF Partitions Disk Size**	**Input RDF Partitions Triples #**
250	32768	800	90 Million	4	8	47 GB	50 Billion
RDF Format to Signatures Conversion Time	**Bloom Counters Construction Time**	**Signature Groups to RDF Format Conversion Time**	**Jena TDB Construction Time**	**RDF-3X Construction Time**	**Total Index Construction Time**		
12764.69	7582.75	1741	8970.4	2364	33422.84		
# of Blooms Constructed	**Bloom Counters Disk Size**	**Jena TDB Index Disk Size**	**RDF-3X Index Disk Size**				
315	27 GB	25 GB	14 GB				

Below are the querying timings. The timings for Candidates Identifying and Index Querying for those candidates are shown separately. We also used different approaches by classifying the query types as Big or Small based on the Number of Triples they contain. All the timings are shown in seconds by default and are an average of three readings taken by clearing out the system cache before each experiment.

Table 9.2.11 Querying Evaluation for Seed Pick Size as 800.

Description:		Querying Evaluation for Seed Pick Size : 800						
Query	**# of Triples in the Query**	**Query Type**	**Approach Used**	**# of Candidate Matches**	**# of False Positives**	**Candidates Identifying Time**	**Index Querying Time**	**Total Querying Time (Identifying + Querying)**
Q1	16	Big	RIS + Jena TDB	1	0	0.08	3.514	3.522
Q2	21	Big	RIS + Jena TDB	1	0	0.107	4.088	4.195
Q3	22	Big	RIS + Jena TDB	3	3	0.133	6.676	6.809
Q4	5	Small	RIS + RDF-3X	315	23	0.074	107.865	107.939

Below is a detailed comparison between the performance of our tool to that of RDF-3X as stand alone for each query. We can observe for all the queries including both big and small type, our approach out performs RDF-3X stand alone approach. Our approach is significantly efficient for bigger queries specifically.

Below is a performance comparison between RIS and RDF-3X. All the timings are shown in seconds by default and are an average of three readings taken by clearing out the system cache before each experiment.

Table 9.2.12 Performance Comparison for Seed Pick Size as 800.

Description:	**Performance Comparison for Seed Pick Size : 800**			
Query	**# of Results**	**Total Querying Time (Identifying + Querying)**	**RDF-3x Only Time**	**Performance Improvement**
Q1	4	3.522	22.261	632%
Q2	10	4.195	3000.922	71535%
Q3	0	6.809	10839.45	159192%
Q4	2481	107.939	216.371	200%

9.2.5 Conclusion for Seed Pick Size

The seed pick size experiments yield very similar results when it comes to indexing time.

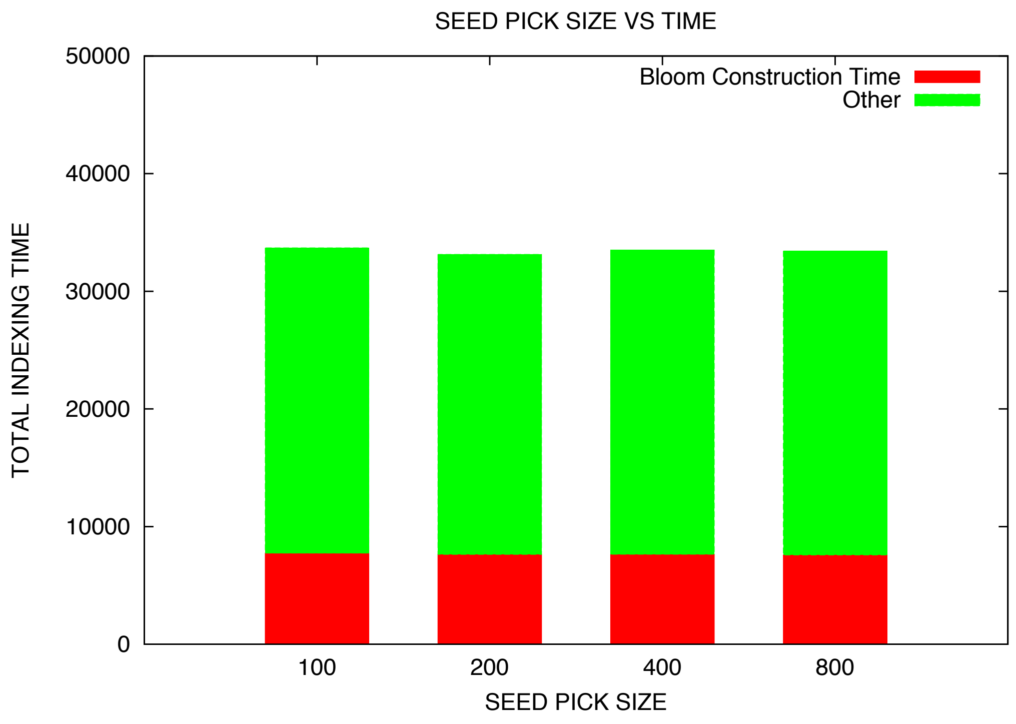


Figure 8.2 A graph between Fanout and Total Indexing Time.

The index building time is almost the same for all the seed pick size values. The bloom construction time was almost the same too.

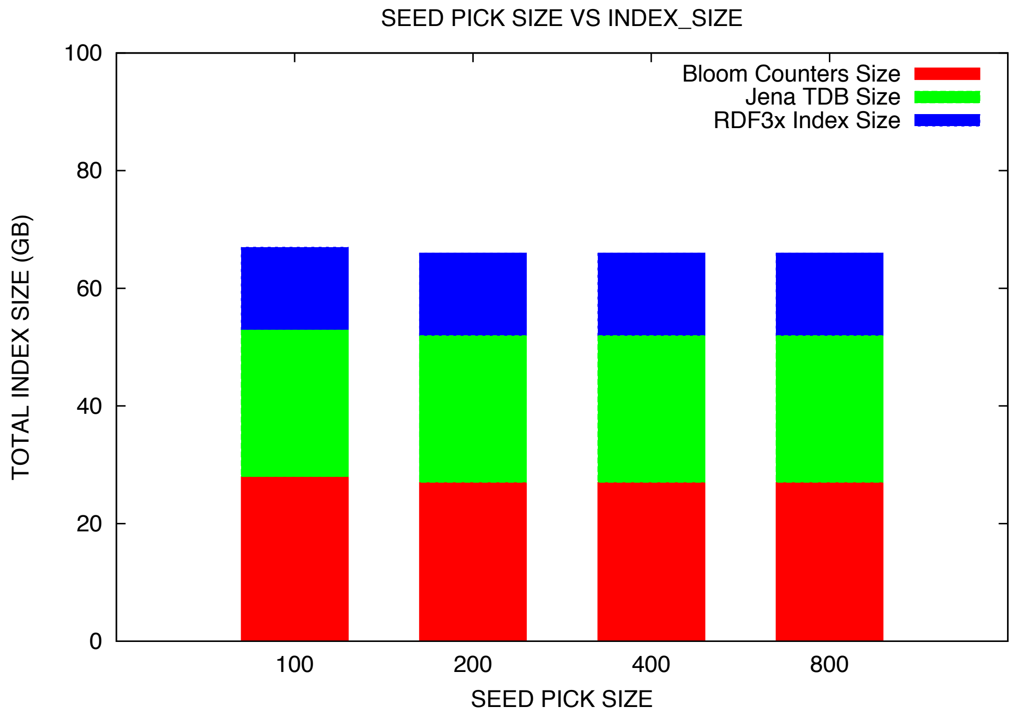


Figure 8.2 A graph between Fanout and Total Indexing Time.

So, it can be deduced form the above graphs that the effect of seed pick size does not have a large influence on our indexing. Now we will see how it effects our querying process.

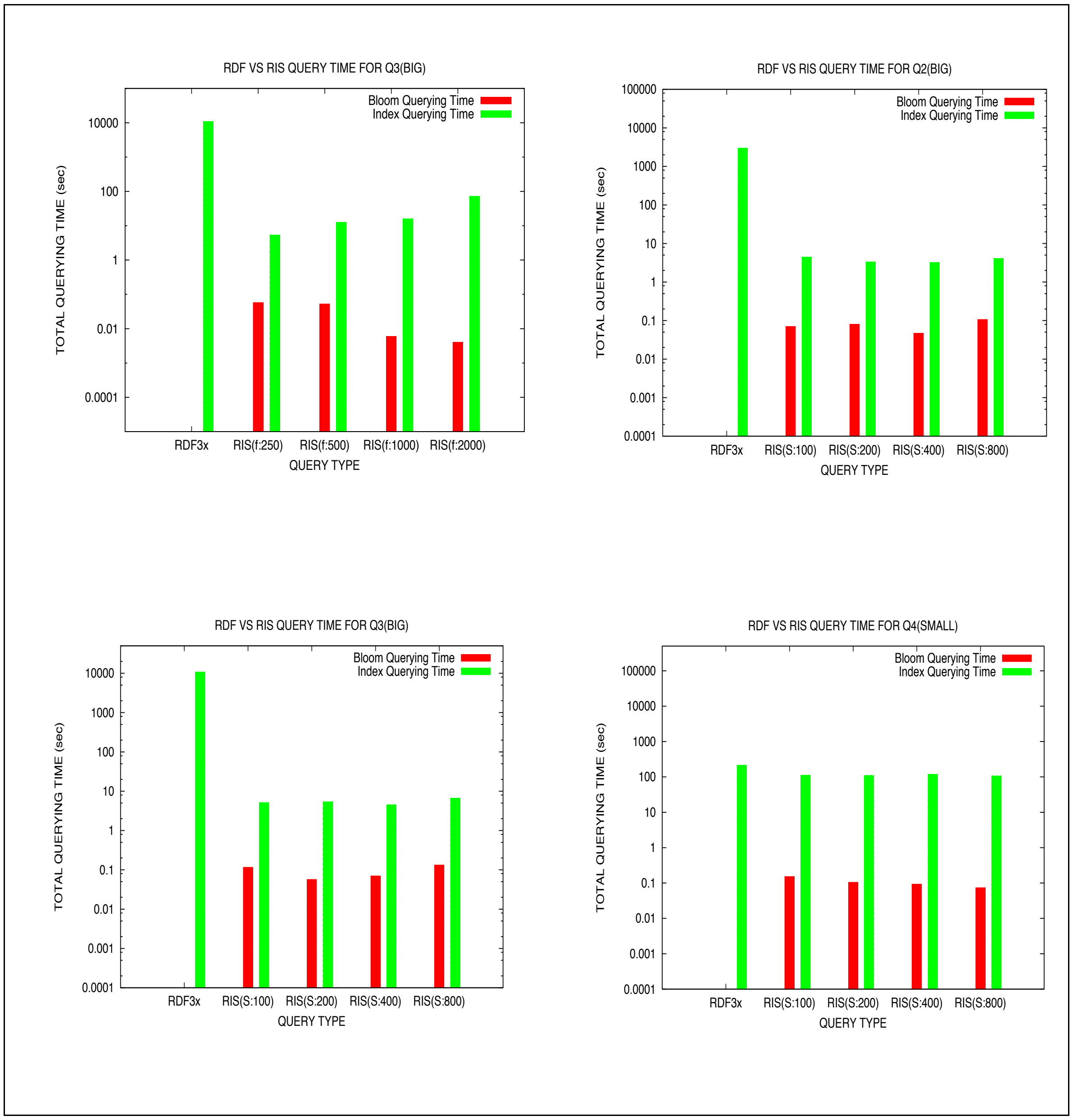


Figure 8.3 Graphs between the Query Type for different Seed Pick Size values and Total Querying Time for queries Q1, Q2, Q3 and Q4.

In almost all the cases we comprehensively beat RDF-3X. But, our performance is at its best in the section 9.2.3. among all. Therefore it is concluded that for a Seed Pick Size of 400 our approach is most optimal.

CHAPTER 10

CONCLUSION

It can be concluded based on the results that our approach does work for both large queries with high selectivity and small queries with low selectivity. We particularly out perform RDF-3x in the case of large queries.

One of the main reasons why our approach works wonderfully well is we do not have to query through the whole large data index for the queries. We use the bit comparison mechanism in Bloom Counters to eliminate the graphs which definitely do not have any potentials results.

Also, we show that when the parameters for Bloom Counters are accordingly provided the percentage of false positives for selectivity queries could be less than 1% of the total data. Thus we save a lot of time by decreasing the false positives too.

REFERENCES

1. M. Garland and D. B. Kirk, “Understanding Throughput-Oriented Architectures,” Commun. of ACM, vol. 53, pp. 58–66, November 2010.
2. S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, “An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS,” in Solid-State Circuits Conference, 2007, pp. 98–589.
3. L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, “Larrabee: A Many-Core x86 Architecture for Visual Computing,” ACM Transactions on Graphics, vol. 27, pp. 18:1–18:15, August 2008.
4. T. Mattson, R. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, “The 48-core SCC Processor: the Programmer’s View,” in Proc. of Intl. Conf. for High Performance Computing, Networking, Storage and Analysis, Nov 2010, pp. 1–11.
5. “Resource Description Framework (RDF),” http://www.w3.org/RDF.
6. “SPARQL Query Language for RDF,” http://www.w3.org/TR/rdf-sparqlquery/.
7. C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann, “DBpedia - A crystallization point for the Web of Data,” Journal of Web Semantics: Science, Services and Agents on the World Wide Web, vol. 7, no. 3, pp. 154–165, September 2009.
8. “Semantic Web Challenge,” http://challenge.semanticweb.org/.

9. P. Gschwandtner, T. Fahringer, and R. Prodan, “Performance Analysis and Benchmarking of the Intel SCC,” in Proc. of Intl. Conf. on Cluster Computing, Sept. 2011, pp. 139–149.

10. K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds, “Efficient RDF Storage and Retrieval in Jena2,” in Proc. of SWDB’03, 2003, pp. 131– 150.

11. S. Harris and N. Gibbins, “3store: Efficient Bulk RDF Storage,” in Practical and Scalable Semantic Systems, 2003.

12. J. Broekstra, A. Kampman, and F. van Harmelen, “Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema,” in Proc. of ISWC ’02, pp. 54–68.

13. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan, “An Efficient SQLBased RDF Querying Scheme,” in Proc. of the 31st VLDB Conference, Trondheim, Norway, 2005, pp. 1216–1227.

14. L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu, “RStar: an RDF storage and query system for enterprise resource management,” in Proc. of CIKM ’04, Washington, D.C., USA, 2004, pp. 484–491.

15. J. J. Levandoski and M. F. Mokbel, “RDF Data-Centric Storage,” in Proc. ICWS ’09, Washington, DC, 2009, pp. 911–918.

16. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, “Scalable Semantic Web Data Management Using Vertical Partitioning,” in Proc. of the 33rd VLDB conference, 2007, pp. 411–422. 17 T. Neumann and G. Weikum, “The RDF-3X

engine for scalable management of RDF data," The VLDB Journal, vol. 19, no. 1, pp. 1–113, 2010.

17. C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," Proc. VLDB Endow., vol. 1, no. 1, pp. 1008–1019, 2008.
18. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix "Bit" Loaded: A Scalable Lightweight Join Query Processor for RDF Data," in Proc. Of the 19th Intl. Conference on World Wide Web, Raleigh, North Carolina, USA, 2010, pp. 41–50.
19. A. Harth, J. Umbrich, A. Hogan, and S. Decker, "YARS2: A Federated Repository for Querying Graph Structured Data From the Web," in Proc. of ISWC'07/ASWC'07, Busan, Korea, 2007, pp. 211–224.
20. S. Harris, N. Lamb, and N. Shadbolt, "4store: The Design and Implementation of a Clustered RDF Store," in Proc. of 5th Intl. Workshop on Scalable Semantic Web Knowledge Base Systems, 2009, pp. 94–109.
21. A Owens, A. Seaborne, N. Gibbins, and M. Schraefel, "Clustered TDB: A Clustered Triple Store for Jena," in Technical Report, Electronics and Computer Science, University of Southampton, 2008.

22. J. Weaver and G. T. Williams, "Scalable RDF Query Processing on Clusters and Supercomputers," in Proc. of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems, 2009.

23. P. Castagna, A. Seaborne, and C. Dollin, “A Parallel Processing Framework for RDF Design and Issues,” HP Labs, Bristol, Tech. Rep., 2009, www.hpl.hp.com/techreports/2009/HPL-2009-346.pdf.

24. R. Sridhar, P. Ravindra, and K. Anyanwu, “RAPID: Enabling Scalable Ad-Hoc Analytics on the Semantic Web,” in Proc. of ISWC ’09, 2009, pp. 715–730.

25. M. F. Husain, J. McGlothlin, M. M. Masud, L. R. Khan, and B. Thuraisingham, “Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing,” IEEE Transactions on Knowledge and Data Engineering, vol. 23, pp. 1312–1327, 2011.

26. J.Weaver and J. A. Hendler, “Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples,” in Proc. of ISWC ’09, Chantilly, VA, 2009, pp. 682–697.

27. J. Urbani, S. Kotoulas, E. Oren, and F. Harmelen, “Scalable Distributed Reasoning Using MapReduce,” in Proc. of ISWC ’09, 2009, pp. 634–649.

28. J. Huang, D. J. Abadi, and K. Ren, “Scalable SPARQL Querying of Large RDF Graphs,” PVLDB, vol. 4, no. 11, pp. 1123–1134, 2011.

29. V. Vidal, S. Vernhes, and G. Infantes, “Parallel AI Planning on the SCC,” in 4rd Many-core Applications Research Community Symposium (MARC 2011), Potsdam, Germany, 2011, pp. 1–6.

30. P. Petrides, A. Diavastos, and P. Trancoso, “Exploring Database Workloads on Future Clustered Many-Core Architectures,” in 3rd Many-core Applications Research Community Symposium (MARC 2011), Ettlingen, Germany, 2011, pp. 81–84.

31. G. Karypis and V. Kumar, “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs,” SIAM Journal on Scientific Computing, vol. 20, pp. 359–392, December 1998.

32. I. A. C. Urena, “RCKMPI User Manual,” Intel Braunschweig, 2011.

33. I. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum, “YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages,” in Proc. of WWW ’11, 2011, pp. 229–232.

34. “Uniprot RDF Distribution,” http://www.uniprot.org/downloads.

35. Y. Guo, Z. Pan, and J. Heflin, “LUBM: A benchmark for OWL knowledge base systems,” Web Semantics: Science, Services and Agents on the World Wide Web, vol. 3, pp. 158–182, October 2005.

36. V. Slavov, P. Rao, D. Barenkala, and S. Paturi, “Towards RDF Query Processing on the Intel SCC,” University of Missouri-Kansas City, Tech. Rep. TR-DB-2012-01, 2012, http://r.web.umkc.edu/raopr/TR-DB-2012-01.pdf.

37. Dipali Pal and Praveen R. Rao. 2011. A tool for fast indexing and querying of graphs. In Proceedings of the 20th international conference companion on World wide web (WWW '11). ACM, New York, NY, USA, 241-244.

VITA

Srivenu Paturi is currently working as a Platform Engineer with a New York based internet company. He got a Bachelor of Technology from Jawaharlal Nehru Technological University from Hyderabad India in 2010.