A NEW APPROACH FOR FAST PROCESSING OF SPARQL QUERIES ON RDF QUADRUPLES

A DISSERTATION IN

Computer Science
and
Telecommunications and Computer Networking

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

DOCTOR OF PHILOSOPHY

by

VASIL GEORGIEV SLAVOV

M.S., University of Missouri-Kansas City, 2012
B.A., William Jewell College, 2005

Kansas City, Missouri
2015

A NEW APPROACH FOR
FAST PROCESSING OF SPARQL QUERIES ON
RDF QUADRUPLES

Vasil Georgiev Slavov, Candidate for the Doctor of Philosophy Degree
University of Missouri-Kansas City, 2015

ABSTRACT

The Resource Description Framework (RDF) is a standard model for representing data on the Web. It enables the interchange and machine processing of data by considering its semantics. While RDF was first proposed with the vision of enabling the Semantic Web, it has now become popular in domain-specific applications and the Web. Through advanced RDF technologies, one can perform semantic reasoning over data and extract knowledge in domains such as healthcare, biopharmaceuticals, defense, and intelligence. Popular approaches like RDF-3X perform poorly on RDF datasets containing billions of triples when the queries are large and complex. This is because of the large number of join operations that must be performed during query processing. Moreover, most of the scalable approaches were designed to operate on RDF triples instead of quads. To address these issues, we propose to develop a new approach for fast and cost-effective processing of SPARQL queries on large RDF datasets containing RDF quadruples (or quads). Our approach employs a *decrease-and-conquer* strategy: Rather than indexing the entire RDF dataset, it

identifies groups of similar RDF graphs and indexes each group separately. During query processing, it uses a novel filtering index to first identify candidate groups that may contain matches for the query. On these candidates, it executes queries using a conventional SPARQL processor to produce the final results. A query optimization strategy using the candidate groups to further improve the query processing performance is also used.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a dissertation titled "A New Approach for Fast Processing of SPARQL Queries on RDF Quadruples," presented by Vasil Georgiev Slavov, candidate for the Doctor of Philosophy degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Praveen R. Rao, Ph.D., Committee Chair
Department of Computer Science Electrical Engineering

Yugyung Lee, Ph.D.
Department of Computer Science Electrical Engineering

Deep Medhi, Ph.D.
Department of Computer Science Electrical Engineering

Vijay Kumar, Ph.D.
Department of Computer Science Electrical Engineering

Appie van de Liefvoort, Ph.D.
Department of Computer Science Electrical Engineering

CONTENTS

ABSTRACT iii

LIST OF ILLUSTRATIONS viii

LIST OF TABLES x

ACKNOWLEDGMENTS xi

Chapter

1. INTRODUCTION 1

2. BACKGROUND AND MOTIVATIONS 4

2.1.Semantic Web 4

2.2.RDF 6

2.3.SPARQL 8

2.4.Related Work 10

2.5.Motivations 13

3. THE DESIGN OF RIQ 15

3.1.Pattern Vectors 16

3.2.Filtering Index 20

3.3.Query Processing 27

4. IMPLEMENTATION OF RIQ 32

4.1.System Architecture 32

4.2.Implementation 36

4.3.Limitations 37

5. EVALUATION 40
5.1.Datasets 41
5.2.Queries 41
5.3.Indexing 42
5.4.Query Processing 45
6. CONCLUSION AND FUTURE WORK 61
Appendix
A. QUERIES 63
A.1LUBM Queries 64
A.2BTC-2012 Queries 75
B. SPARQL GRAMMAR 85
REFERENCES 87
VITA 93

ILLUSTRATIONS

Figure Page

1. Linking Open Data cloud 5
2. The Semantic Web stack 7
3. An example of a SPARQL query 8
4. Big picture of RIQ 15
5. Pattern Vectors of RIQ 16
6. Grouping of similar PVs in RIQ 22
7. Grouping five PVs into two connected components 23
8. LSH: probability vs similarity 24
9. Bloom filter test operation 25
10. An example of a BGP Tree 29
11. The BGP Tree after pruning 29
12. Architecture of RIQ 32
13. Filtering Index generation in RIQ 33
14. Query execution in RIQ 35
15. Query processing times for LUBM, large queries 56
16. Query processing times for BTC-2012, large queries 57
17. Query processing times for LUBM, small queries 58
18. Query processing times for BTC-2012, small queries 59
19. Query processing times for BTC-2012, multi-BGP queries 60

20. Visual representation of LUBM query L1 with a large, complex BGP . . 66
21. Visual representation of LUBM query L2 with a large, complex BGP . . 67
22. Visual representation of LUBM query L3 with a large, complex BGP . . 69
23. Visual representation of BTC query B1 with a large, complex BGP . . . 76
24. Visual representation of BTC query B2 with a large, complex BGP . . . 78

TABLES

Table Page

1. Transformations in RIQ 17
2. Queries for LUBM and BTC-2012. 43
3. RIQ's indexing performance: construction time 45
4. RIQ's indexing performance: settings and size 45
5. Filtering time for RIQ on LUBM. 46
6. Filtering time for RIQ on BTC-2012. 46
7. Fastest approach for processing queries over LUBM. 50
8. Fastest approach for processing queries over BTC-2012. 50
9. Geometric Mean of the query processing times for LUBM 51
10. Geometric Mean of the query processing times for BTC-2012 51
11. Query processing times for LUBM 52
12. Query processing times of LSH and Jaccard using RIQ for LUBM 53
13. Query processing times for BTC-2012. 54
14. Query processing times of LSH and Jaccard using RIQ for BTC-2012 . . 54
15. Query processing times for multi-BGP BTC-2012 queries. 55

ACKNOWLEDGMENTS

I would like to thank my dissertation advisor Dr. Praveen Rao. Without his guidance, support, motivation and extreme patience, this work would not have been possible. My experience in working with him on this and other research projects inspired and motivated me to continue my academic career after completing my MS degree and pursue a Ph.D. degree in Computer Science. Dr. Rao supported me financially which allowed me to dedicate all of my time to research. I would also like to to thank Anas Katib for the very valuable contributions he made to this project and all the work he did as a co-author for the paper submissions on this topic. Finally, I want to thank my family for their support throughout the years I have been in school.

This work was supported by the National Science Foundation under Grant No. 1115871.

CHAPTER 1

INTRODUCTION

The Resource Description Framework (RDF) has become a widely used, standardized model for publishing and exchanging data [10]. From research institutions and government agencies [1, 2], to large media companies [41, 8] and retailers [4], RDF has been adopted and used in production environments at a scale which was only dreamed of in the initial stages of the development of the Semantic Web vision of Tim Berners-Lee [12].

The most popular use case for RDF is Linked Data, a set of technologies and specifications for publishing and interconnecting structures data on the Web in a machine-readable and consumable way [24].

In addition to the increased popularity of RDF technologies, a number of very large RDF datasets (*e.g.* Billion Triples Challenge (BTC) [11] and Linking Open Government Data (LOGD) [6]) have pushed the limits of scalability in terms of indexing and query processing. Both BTC and LOGD are non-synthetic datasets which have long surpassed the billion-quad mark and contain millions of RDF graphs.

Researchers in both the Database and the Semantic Web communities have proposed scalable solutions for processing large RDF datasets [17, 59, 47, 19, 37, 25, 63, 64], but all of those approaches have been evaluated using datasets containing RDF triples. The RDF quadruples contained in more recent, larger datasets extend the idea of the RDF triple by adding a context/graph name to the traditional subject,

predicate, object statement. The context names the graph to which the triple belongs and enables the use of the SPARQL's 1.1 `GRAPH` keyword [13] to match a specific graph pattern within a single RDF graph among millions of individual graphs in a large RDF dataset. Simply ignoring the context of a quad is not possible, as we show in the following chapters, because it yields incorrect results.

In addition to not supporting RDF quadruples, none of the state-of-the-art solutions have been evaluated using SPARQL queries containing large, complex graph patterns (*e.g.* with a large number of triple patterns and/or undirected cycles). In fact, in the following pages, we show that those approaches yield very poor performance on SPARQL queries with large, complex graph patterns. One of the reasons for the poor performance is the large number of join operations which must be done during query processing. In the evaluation we performed, any approach which first finds matches for subpatterns in a large graph pattern and then merges partial results with join operations does not scale to billion-quad datasets.

We propose a new approach for fast processing of SPARQL queries on RDF quads called RIQ (**R**DF **I**ndexing on **Q**uadruples). The contributions of this work are:

- A new vector approach for representing RDF graphs and SPARQL queries which captures the properties of both triples in a graph and triple patterns in a query.
- A new filtering index which groups similar RDF graphs using Locality Sensitive Hashing [38] and a combination of Bloom Filters and Counting Bloom Filters [28] for compact storage.
- A new *decrease & conquer* approach for query processing which uses the

filtering index to process SPARQL queries efficiently by identifying candidate groups of RDF graphs which may contain a match for a query without false dismissals.

- A comprehensive evaluation of RIQ on a synthetic and a real RDF dataset containing about 1.4 billion quads each with a large number of queries and compared against state-of-the-art approaches such as RDF-3X [46], Jena TDB [18] and Virtuoso [31].

The rest of this work is organized as follows. Chapter 2 provides the background on Semantic Web focusing on RDF and SPARQL. It also describes the related work and the motivation for our work. Chapter 3 describes the design of RIQ, starting with an in-depth introduction to the new vector representation of RDF graphs and continuing with the generation of a filtering index and the role it plays during query processing. Chapter 4 covers the implementation of RIQ and highlights some of the limitations of this approach. Chapter 5 reports the results of the comprehensive evaluation of RIQ. Finally, we conclude in Chapter 6.

CHAPTER 2

BACKGROUND AND MOTIVATIONS

2.1 Semantic Web

The term Semantic Web was invented by Tim Berners-Lee, the inventor of the World Wide Web. In simple terms, the Semantic Web refers to the web of data which can be processed by machines [57]. The W3C, the standard organization behind the WWW, the Semantic Web and many other web technologies, defines it as a common framework designed for data sharing and re-usability across applications and enterprises [16]. There has been some confusion regarding the relationship between the Semantic Web and another term closely related to it, Linked Data. The most common view is that the "Semantic Web is made up of Linked Data" [5] and that "the Semantic Web is the whole, while Linked Data is the parts" [5]. According to Tim Berners-Lee, Linked Data is "the Semantic Web done right" [22].

Linked Data refers to the set of technologies and specifications or best practices used for publishing and interconnecting structured data on the Web in a machine-readable and consumable way [24]. The main technologies behind Linked Data are URIs, HTTP, RDF and SPARQL. URIs are used as names for things. HTTP URIs are used for looking up things by machines and people (retrieving resources and their descriptions). RDF is the data model format of choice. SPARQL is the standard query language. In essence, Linked Data enables connecting related data across the Web using URIs, HTTP and RDF.

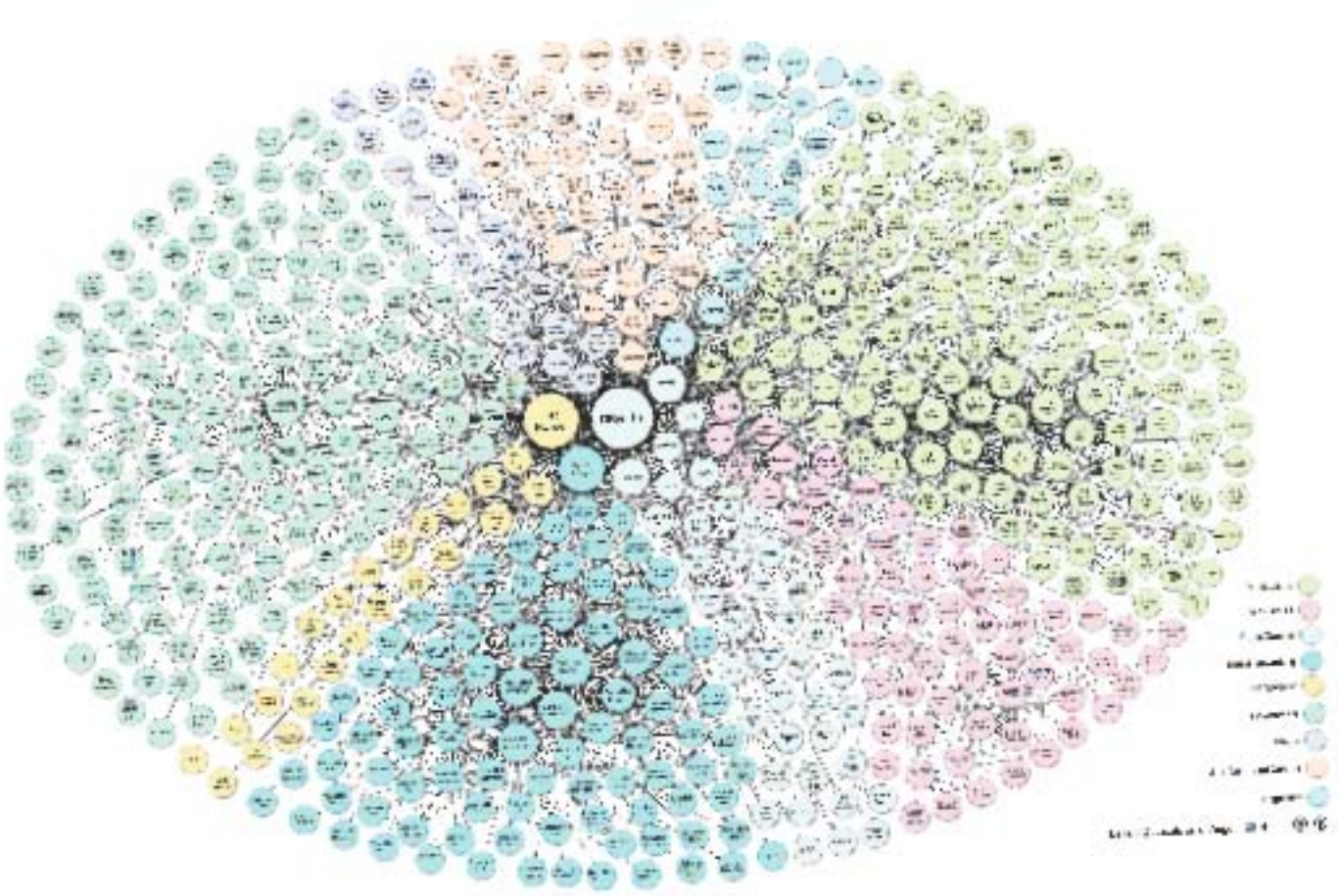


Figure 1. Linking Open Data cloud
(Source: by Richard Cyganiak and Anja Jentzsch `http://lod-cloud.net/`)

Linked Data is a popular use case of RDF on the Web; it has a large collection of different knowledge bases, which are represented in RDF (*e.g.*, DBpedia, Data.gov [6]). With a growing number of new applications relying on Semantic Web technologies (*e.g.*, Pfizer [9], Newsweek, BBC, The New York Times, Best Buy [4]) and the availability of large RDF datasets (e.g., Billion Triples Challenge (BTC) [11], Linking Open Government Data (LOGD) [6]), there is a need to advance the state-of-the-art in storing, indexing, and query processing of RDF datasets. The Linking Open Data (LOD) [42] cloud diagram shown in Figure 1 visualizes the datasets which have been published in Linked Data format. As of this writing, there were 570 datasets published. The size of the circles corresponds to the size of the datasets and the largest circles indicate datasets which exceed 1 billion triples.

The Semantic Web encompasses a number of different technologies and specifications. Figure 2 identifies most of them: they range from low-level hypertext web technologies (IRI, Unicode, XML), through standardized Semantic Web technologies (RDF, RDFS, OWL, SPARQL), to unrealized Semantic Web technologies (cryptography, user interfaces) [15]. In this work, we are going to focus on the middle layer: the standardized Semantic Web technologies RDF [10] and SPARQL [14].

2.2 RDF

The Resource Description Framework (RDF) is a widely used model for data interchange which has been standardized into a set of W3C specifications [10]. RDF is intended for describing web resources and the relationships between them. RDF expressions are in the form of subject-predicate-object statements. These statements are also known as (s, p, o) tuples or more shortly, triples. RDF triples belong to the

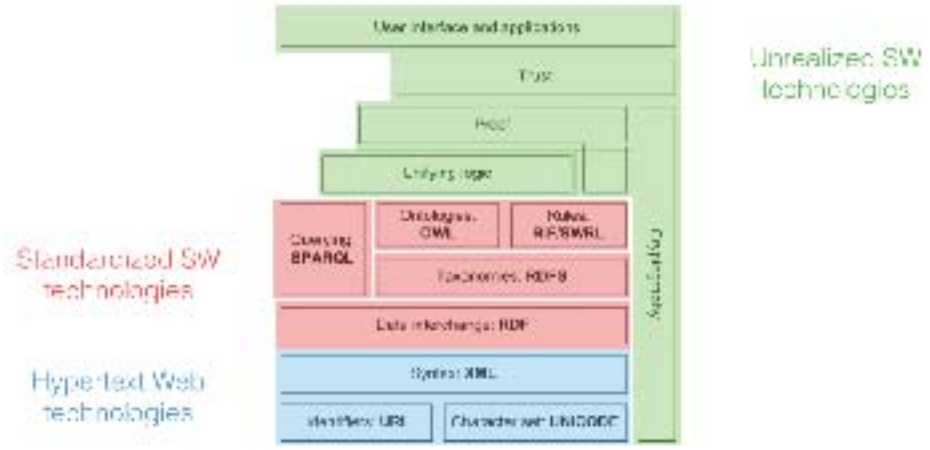


Figure 2. The Semantic Web stack
(Source: `http://en.wikipedia.org/wiki/Semantic_Web_Stack`)

set:

$$(U \cup B) \times U \times (U \cup L \cup B)$$

where U are URIs, L are literals and B are blank nodes and all three are disjoint sets. An RDF *term* is $U \cup L \cup B$. An RDF *element* is any subject, predicate or object [40].

Blank nodes are used to identify unknown constants. Blank nodes are useful for making assertions where something is an object of one statement and a subject of another. For example, the director of X is Godard and the *year* of X is *1970*. A query can be issued for what Godard directed in 1970 and X will be the result.

Resources are uniquely identified using URIs (Uniform Resource Identifiers). Resources are described in terms of their properties and values. A group of RDF statements can be visualized as a directed, labeled graph. The source node is the

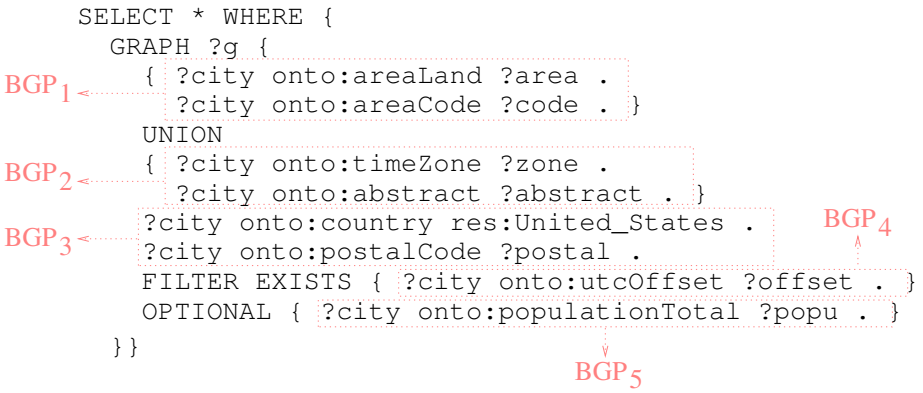


Figure 3. An example of a SPARQL query

subject, the sink node is the object, and the predicate/property is the edge. Quads extend the idea of triples by adding a fourth entity called a context. The context names the graph to which the triple belongs. Triples with the same context belong to the same graph. RDF data my be serialized in a number of different formats, but the most common ones are RDF/XML, Notation-3 (N3), Turtle (TTL), NTriples (NT), NQuads (NQ).

2.3 SPARQL

SPARQL is the standard query language for RDF data. The fundamental operation in RDF query processing is *Basic Graph Pattern Matching* [13]. A Basic Graph Pattern (BGP) in a query combines a set of triple patterns. BGP queries are a conjunctive fragment which expresses the core Select-Project-Join paradigm in database queries [40]. The SPARQL triple patterns (s, p, o) are from the set:

$$(U \cup B \cup V) \times (U \cup V) \times (U \cup L \cup B \cup V)$$

where U are URIs, L are literals, B are blank nodes, and V are variables [40]. The variables in the BGP are bound to RDF terms in the data during query processing via subgraph matching [13]. Join operations are denoted by using common variables in different triple patterns. SPARQL's `GRAPH` keyword [13] can be used to perform BGP matching within a specific graph (by naming it) or in any graph (by using a variable for the graph name). `GRAPH` queries are used on RDF quadruples which contain the context/graph to which a particular triple belongs. Figure 3 shows an example of such a query.

There are three main types of SPARQL queries. The first type is star-join queries. Those queries have the same variable in the subject position (*e.g.* `?`x `type :artist .` `?`x `:firstName ?`y `.` `?`x `:paints ?`z `.`). The second most common type are the path or chain queries where the object of each pattern is joined with the subject of the next one (*e.g.* `?`x `:paints ?`y`.` `?`y `:exhibitedIn ?`z `.` `?`z `:locatedIn :madrid .`) [40]. Finally, more complex SPARQL queries which can be typical for real-world usage, may contain undirected cycles [53] (*e.g.* `?`a `:p ?`b `.` `?`b `:q ?`c `.` `?`a `:r ?`c `.`).

In terms of selectivity and join results, queries may be categorized in three different groups [19]. The first group is queries which contain highly selective triple patterns (*e.g.* `?`s `:residesIn USA .` `?`s `:hasSSN "123-45-6789" .`). The second group consists of queries with triple patterns with low-selectivity, but which generate few results. Those queries are also known as highly selective join queries (`?`s `:residesIn India .` `?`s `:worksFor BigOrg .`). Finally the third group of SPARQL queries contains low-selectivity triple patterns and low-selectivity join re-

sults (*e.g.* ?s :residesIn USA . ?s :hasSSN ?y .) [19].

2.4 Related Work

Since the early vision of the Semantic Web in 2001 [12], many approaches have been developed for indexing and querying RDF data. Overall, there are two storage and query processing categories of RDF solutions: centralized and distributed. Because RIQ is a centralized approach, we are only going to focus on this category. Within the centralized solutions, there are four major groups of applications: triple stores, vertically partitioned tables, property tables, and specialized systems [36]. Some of these systems have originated in the Semantic Web community such as Jena [62] and Sesame [29], others have started in the database community such as SW-Store [17], RDF-3X [46], and Hexastore [59].

2.4.1 Triple stores

Triple stores store RDF triples in a three-column table. Additional indexes and statistics are also maintained. Entities from the triples (constants, properties, resource identifiers) are converted to numerical ids. The most prominent examples of triple store RDF solutions are RDF-3X [46], Hexastore [59], and DB2RDF [25] from the database community and 3store [33], and Virtuoso [31] from the Semantic Web community.

RDF-3X is one of the most popular centralized RDF processing research applications [46, 47, 48]. While RDF-3X has been able to scale remarkably well throughout the years considering that active development has stopped, it is missing support for some more recent RDF and SPARQL specifications. The most glaring omission is

support for quads. While RDF-3X is considered a triples store, it never actually maintains a single triples table, but rather builds clustered B+ tree indexes on all six combinations of subject-predicate-object triples. RDF-3X also makes use of six additional aggregate indexes, one for each possible pair of (s, p, o) triples and each order. Furthermore, RDF-3X is able to keep the size of the index small by encoding strings into ids and by compressing the leaves of the B++ tree indexes in pages [36]. At query time, RDF-3X uses a new join ordering method based on selectivity estimates based on statistics it maintains.

Virtuoso employs space-optimized mapping to identifiers. Only long IRIs as converted to IDs, otherwise the text of each RDF term is stored. Virtuoso creates a GSPO quad table with a primary index on all four (graph, subject, predicate, object) attributes. Virtuoso also creates a bitmap OGPS index. It builds a bit-vector of each OGP combination where the bit position for each subject is set to one for each OGP bit-vector in the data. In order to make its indexes more compact, Virtuoso eliminates common prefixes and strings. Further, it compresses each page [31].

During query processing, Virtuoso resolves simple joins by calculating the conjunction of sparse bit vectors by finding their intersection. Virtuoso does not use pre-calculated statistics for cost estimation. It employs query-time sampling and makes these estimates on the fly [36].

2.4.2 Vertically Partitioned Tables

Vertically partitioned tables maintain one table for each property. They exploit the fact that most RDF datasets contain a fixed number of properties. Vertically partitioned tables are also known as horizontal (binary) table stores [51]. The columns

in each property table are limited only to subjects and objects, and an optional graph id if RDF quadruples are being stored. Column stores use ids for each column entry and reconstruct the row at query time. While vertically partitioned tables are efficient for triple patterns with fixed properties, they are very expensive for triple patterns with a property variable because of the need to access all 2-column tables. [36]. These very narrow tables can then be stored in a column store. Because the entries in each column have the same type, the tables can be efficiently compressed. The most popular example of a vertically partitioned table is SW-Store [17].

2.4.3 Property Tables

Property tables exploit the fact that a large number of subjects have the same predicate. Subjects with similar predicates are grouped together in the same table. The goal is to reduce self-joins. The effect is that queries with those properties are more efficient, but extra work is required for subjects without those properties or subjects with many objects for the same property. Two examples of this approach are Oracle's RDF storage implementation [30] and the Apache Jena project [61]. Apache Jena TDB is the persistent graph storage layer for Jena. The first edition of Jena was a SQL-based system. Jena TDB exploits memory-mapped I/O on 64-bit hardware instead of using a custom caching mechanism [49]. Jena TDB creates three composite B+ tree indexes: SPO, POS, OSP. It does not create a triple table; instead, all three indexes contain a subject, a predicate, and an object. On one hand, this avoids full table scans at query time when doing triple matching. But on the other, this increases loading time significantly. The evaluation Chapter 5 shows this when comparing indexing times of RDF-3X and Jena TDB. Jena uses 64-bit node ids which

serve as references in a node table created during indexing. Part of Jena's claim to fame is the significant performance improvements it is able to achieve compared to its competitors when reading and/or writing the translation node table [49].

BGP matching in Jena TDB is done by choosing the most matching index. If the query triple pattern has a known S and P, the SPO index is used. Next, a range scan is performed to find the node ids of the unknown triple pattern terms (the variables). The node id to RDF term is left for the very last step and is only performed on actual result triples.

2.4.4 Specialized and Distributed Systems

Finally, a number of diverse and specialized systems use their own, unique techniques for storing RDF data. BitMat [19] manages to process queries with low selectivity triple patterns which produce large intermediate join results quickly by performing in-memory processing of bit matrices. Others exploit the graph properties of RDF data [58, 26, 65]. Unfortunately, these techniques have not been tested against larger RDF datasets of more than 50 million triples. Distributed/parallel RDF query processing has gained popularity in more recent research [37, 64], however, our approach focuses on localized and centralized RDF query processing.

2.5 Motivations

We motivate our work with two key observations. The first one is that the approaches outlined in the related work above were designed for processing of RDF triples, not quadruples. While it is possible to ignore the context in a quad, it is certainly not desirable because the likelihood of incorrect results is very high due to

bindings to a BGP from different RDF graphs. For example, let us consider two quads which belong to two different graphs: `<a> <b> <c> <g1> . <a> <b> <e> <g2>` . If we index these quads with an approach which supports only triples, such as RDF-3X, and issue the following query `SELECT ?x WHERE { GRAPH ?g { ?x <b> <c> . ?x <b> <e> . } }` , we would get `<a>` as the result. However, a quad store will return an empty result set which is the correct result as there is no single graph which contains both triple patterns (there are two graphs containing partial matches). Because the triple store approach ignores the name of the graph, it treats both triple patterns as if they belong to the same graph.

The second motivation for this work is the relatively simple queries used in the evaluations of the approaches outlined above. The queries used usually contain few triple patterns (no more than 8) and are either star-shaped or contain very short paths/chains. None of the state-of-the-art techniques for RDF query processing have investigated processing SPARQL queries with large and complex BGPs. For example, queries containing undirected cycles such as this one: ?a `<p>` ?b . ?b `<q>` ?c . ?a `<r>` ?c . have not been evaluated. We believe that those types of queries are more realistic for real-world SPARQL usage as they more closely mirror the complex SQL queries used in the relational world.

CHAPTER 3

THE DESIGN OF RIQ

In this section, we are going to describe the design of RIQ and its components. We are going to discuss in detail the way RIQ performs indexing of RDF data and query processing of SPARQL queries. We are going to focus on the novel vector approach for representing RDF graphs and queries, the new filtering index contribution which is at the heart of RIQ, and we are going to conclude with the new *decrease & conquer* approach which RIQ takes during query processing.

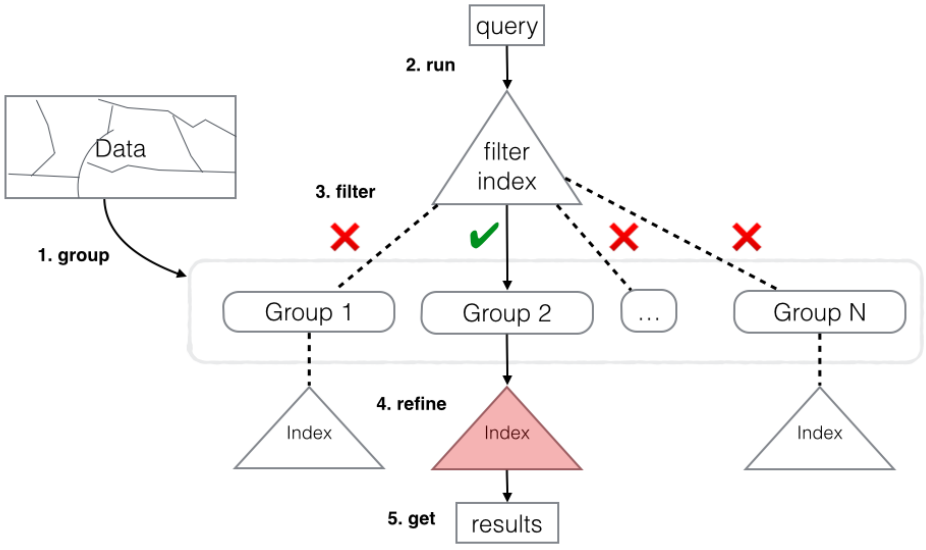


Figure 4. Big picture of RIQ

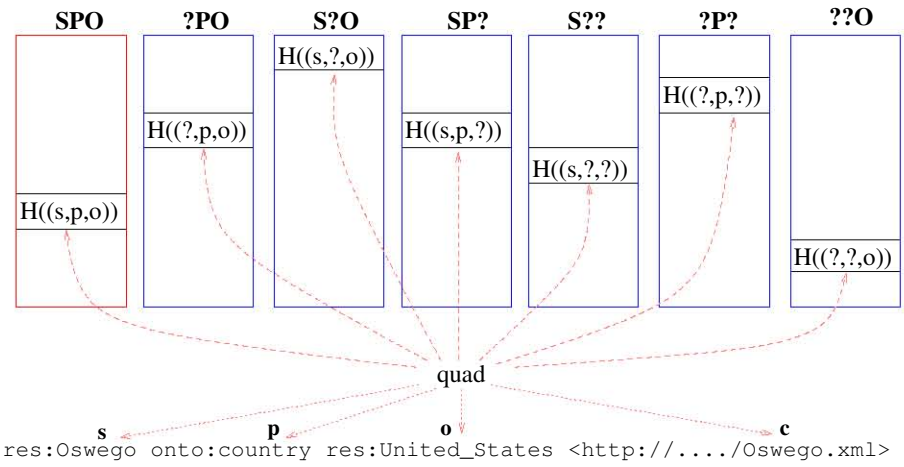


Figure 5. Pattern Vectors of RIQ

3.1 Pattern Vectors

One of the main contributions of this work is a new vector representation of RDF graphs and BGPs. We call this new representation Pattern Vectors (or PVs for short). Pattern Vectors allow us to capture the properties of triples in RDF data and triple patterns in SPARQL query BGPs. Pattern Vectors are an essential part of RIQ and play a central role in all phases of RDF indexing and query processing.

Before describing the design and use of Pattern Vectors, let us define some prerequisite essential transformations of RDF graphs and BGP patterns. First, let's define the set of canonical patterns as $\mathbb{P} = \{SPO,\ SP?,\ S?O,\ ?PO,\ S??,\ ?P?,\ ??O\}$. The set $\mathbb{P}$ represents all the possible permutations of the subject, the predicate, and the object, or their substitution by a variable in a triple or a triple pattern. The transformation of a triple, we denote as $f_D : \mathbb{P} \times \{(s,p,o)\} \rightarrow O_D$. We show the range, O_D, in Table 1. Note that O_D resembles a BGP triple pattern with excluded

Table 1. Transformations in RIQ

Transformation f_D	Transformation f_Q
f_D(SPO, (s,p,o)) = (s,p,o)	f_Q('s p o') = (SPO,(s,p,o))
f_D(SP?, (s,p,o)) = (s,p,?)	f_Q('s p ?v_o') = (SP?,(s,p,?))
f_D(S?O, (s,p,o)) = (s,?,o)	f_Q('s ?v_p o') = (S?O,(s,?,o))
f_D(?PO, (s,p,o)) = (?,p,o)	f_Q('?v_s p o') = (?PO,(?,p,o))
f_D(S??, (s,p,o)) = (s,?,?)	f_Q('s ?v_p ?o') = (S??,(s,?,?))
f_D(?P?, (s,p,o)) = (?,p,?)	f_Q('?v_s p ?v_o') = (?P?,(?,p,?))
f_D(??O, (s,p,o)) = (?,?,o)	f_Q('?v_s ?v_p o') = (??O,(?,?,o))

variable names.

Next, we denote the transformation of a BGP triple pattern as $f_Q : T \rightarrow \mathbb{P}\times O_Q$ where T denotes the set of triple patterns which may appear in a SPARQL query. We show the range, $\mathbb{P} \times O_Q$, in Table 1 which denotes the canonical pattern of a given BGP triple pattern. Note that f_Q('s p o') is a valid transformation (even though it has no variables) because `SELECT ?g WHERE { GRAPH ?g { s p o .} }` is a valid SPARQL 1.1 query.

The purpose of both transformations is to allow us to map both the RDF data and the SPARQL queries to a common reference point which will let us perform different operations on both. In particular, we will be able to test *if a triple is a match for a BGP triple pattern.*

Having defined the essential transformations, let us introduce Pattern Vectors. Given an RDF graph which consists of triples with the same context c, we create a vector representation of the graph which we call a Pattern Vector (PV). We denote this PV as $\overline{V_c}$. Pattern Vectors consist of 7 separate vectors, one for each $r \in \mathbb{P}$: $\overline{V_c}$

Algorithm 1 Construction of the PV of an RDF graph

Require: An RDF graph G with context c
Ensure: PV $\overline{V_c}$

```
for each (s, p, o, c) ∈ G do
    for each r ∈ ℙ do
        insert ℍ(f_D(r, (s, p, o))) into V_{c,r}
for each r ∈ ℙ do
    sort V_{c,r}
return V̄_c
```

$= (V_{c,SPO}, V_{c,SP?}, V_{c,S?O}, V_{c,?PO}, V_{c,S??}, V_{c,?P?}, V_{c,??O})$. Figure 5 shows an example of a Pattern Vector and the mapping between a triple and the 7 vectors in the PV. The individual vectors $V_{c,r}$ consist of the non-negative integer output of the hash function $\mathbb{H} : B \rightarrow \mathbb{Z}^*$ where B is the original triple represented as a bit string. We compute $\mathbb{H}(f_D(r, (s, p, o)))$ for each $r \in \mathbb{P}$. The required space for each $\overline{V_c}$ is linear to the number of quads in the graph. Algorithm 1 describes the construction of the Pattern Vector of an RDF graph.

We base the hash function $\mathbb{H}$ on Rabin's fingerprinting technique [50]. Rabin's fingerprinting technique is efficient to compute and the probability for collision is very low. The hash values we generate are 32-bit unsigned integers. The irreducible polynomial is of degree 31. Given these values, the probability of collision is 2^{-20} [27]. Because all the triples in one RDF graph are unique (by design), the vector $V_{c,SPO}$ is a set of the non-negative integer outputs of $\mathbb{H}$. However, when we start substituting the elements of each triple with variables, the bit string inputs to $\mathbb{H}$ are no longer unique and therefore, the remaining $\overline{V_c}$ vectors are multisets.

We map BGPs to Pattern Vectors in a similar but slightly different way. Start-

Algorithm 2 Construction of the PV of a BGP

Require: A BGP q
Ensure: TPV $\overline{V_q}$
1: **for** each triple pattern $t \in Q$ **do**
2: $(r, o_q) \leftarrow f_Q(t)$
3: insert $\mathbb{H}(o_q)$ into $V_q[r]$
4: **return** $\overline{V_q}$

ing with an empty $V_{q,r}$, we compute $f_Q(t)$ for each triple pattern t and produce a pair (r, o) where r is the canonical pattern for t and o is the input bit string for $\mathbb{H}$. We insert the output of $\mathbb{H}$ into $V_{q,r}$. Similarly to the Pattern Vector generation for RDF graphs, $\overline{V_q}$ consists of one set ($V_{q,SPO}$) and six multisets because different triple patterns containing variables may hash to the same value. While we ignore the `?s ?p ?o` triple pattern during PV construction, we consider it during query processing when we execute the optimized queries. Algorithm 2 describes the construction of the Pattern Vector of a BGP.

In order to accomplish the goal of grouping similar RDF graphs together, we define two operations on Pattern Vectors which let us group similar PVs together. (Grouping of similar RDF graphs helps us quickly identify candidate RDF graphs at query time.) The definitions of the Union and Similarity operations follow:

DEFINITION 3.1 (Union). Given two PVs, say $\overline{V_a}$ and $\overline{V_b}$, their union $\overline{V_a} \cup \overline{V_b}$ is a PV say $\overline{V_c}$, where $V_{c,r} \leftarrow V_{a,r} \cup V_{b,r}$ and $r \in \mathbb{P}$.

DEFINITION 3.2 (Similarity). Given two PVs, say $\overline{V_a}$ and $\overline{V_b}$, their similarity is denoted by $sim(\overline{V_a}, \overline{V_b}) = \max_{r \in \mathbb{P}} sim(V_{a,r}, V_{b,r})$, where $sim(V_{a,r}, V_{b,r}) = \frac{|V_{a,r} \cap V_{b,r}|}{|V_{a,r} \cup V_{b,r}|}$.

3.2 Filtering Index

As described in the previous section, both RDF graphs and BGPs are mapped to their respective Pattern Vectors. This enables us to perform operations such as the Union and Similarity operations defined above. A key necessary condition which connects the two types of Pattern Vectors and is at the heart of RIQ's indexing and query processing is defined and proved below. This condition is concerned with processing BGPs via subgraph matching at query time.

THEOREM 3.3. Suppose $\overline{V_c}$ and $\overline{V_q}$ denote the PVs of an RDF graph and a BGP, respectively. If the BGP has a subgraph match in the RDF graph, then $\bigwedge_{r \in \mathbb{P}} (V_{q,r} \subseteq V_{c,r}) = \texttt{TRUE}$.

We assume that the BGP q denotes a connected graph. Because q has a subgraph match in the graph, every triple pattern in q has a matching triple in the graph. Consider a triple pattern t in q. Let $(r, o) \leftarrow f_Q(t)$. During the construction of V_q, we inserted $\mathbb{H}(o)$ into $V_{q,r}$. Suppose d denotes the matching triple pattern for t in the graph. During the construction of V_c, we had inserted $\mathbb{H}(f_D(r, d))$ into $V_{c,r}$. Also, $\mathbb{H}(o) = \mathbb{H}(f_D(r, d))$. Therefore, elements in $V_{q,r}$ have a one-to-one correspondence with a subset of elements in $V_{c,r}$. Hence, $V_{q,r} \subseteq V_{c,r}$. This is true for every $r \in \mathbb{P}$, and hence, $\bigwedge_{r \in \mathbb{P}} (V_{q,r} \subseteq V_{c,r}) = \texttt{TRUE}$.

Theorem 3.3 enables us to identify the RDF graphs for which a BGP satisfies the necessary condition. Those graphs represent a superset of the actual graphs which contain a subgraph match for the BGP. Since the candidate graphs are a superset, there will be no false dismissals.

In order to eliminate the need for exhaustively testing each Pattern Vector for every BGP during query processing, we have developed a novel filtering index called PV-Index for organizing and processing the millions of Pattern Vectors which represent the RDF graphs. The goal of the PV-Index is to quickly and efficiently identify the candidate RDF graphs. To speed up query time, the PV-Index's role is to discard a large number of RDF graphs without false dismissals.

There are two challenges in developing the PV-Index. The first one is grouping of Pattern Vectors in such a way that discarding of non-matching PVs is possible. The second is minimizing the query processing I/O by designing the PV-Index to store the PVs compactly. The first issue we solve by using Locality Sensitive Hashing (LSH) [38]. LSH has been employed for similarity on sets based on the Jaccard index [35]. Given a set S, $\mathsf{LSH}_{k,l,m}(S)$ is computed by picking $k \times l$ random linear hash functions. The functions are the of the form $h(x) = (ax + b)\ mod\ u$ where u is a prime and a and b are integers such that $0 < a < u$ and $0 \leq b < u$. Next, we go over all the items of the set S and compute $g(S) = \min\{h(x)\}$. Using Rabin's fingerprinting technique [50], we hash each group of l hash values to the range $[0, m-1]$ and we get k hash values for the set S. The main reason for using LSH on sets is that given two sets S_1 and S_2, $p = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$, $\Pr[g(S_1) = g(S_2)] = p$. Furthermore, the probability that $\mathsf{LSH}_{k,l,m}(S_1)$ and $\mathsf{LSH}_{k,l,m}(S_2)$ have at least one identical hash value is $1 - (1 - p^l)^k$. Note that all these properties are valid for multisets as well. Figure 8 shows the way the probability for collision changes based on the similarity of the input for different value of k and l.

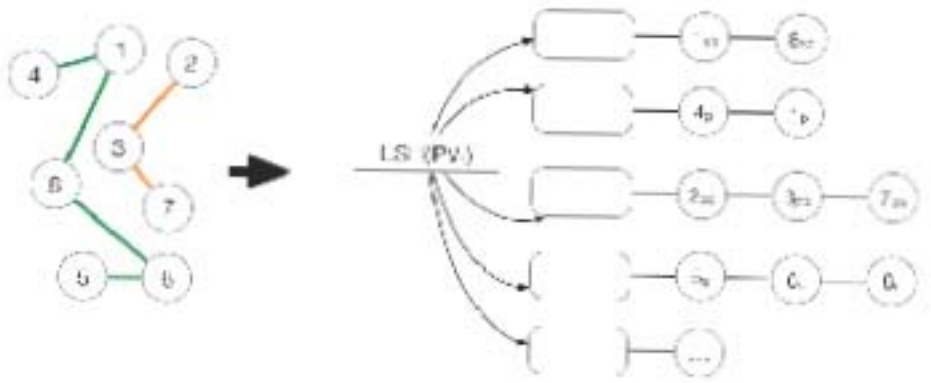


Figure 6. Grouping of similar PVs in RIQ

In order to compactly represent the PV-Index, we make use of Bloom filters (BFs) and Counting Bloom Filters (CBFs) [28]. Bloom filters are a probabilistic data structure which is used for representing a set of items and for testing the membership in that set. They support two operations: add and test. Bloom filters are usually implemented as bit vectors and k different hash functions are used for adding an element to the set. Counting Bloom filters use n-bit counters instead. The test operation's output is either definitely not in the set or may be is in the set. The probability for false positives is approximately $(1 - e^{-kn/m})^k$ where m is the length of the bit vector, k is the number of hash functions and n is the number of items added to the Bloom filter. It is not possible to remove items from the set because that would introduce false negatives. Given a predetermined capacity of a Bloom filter (the number of elements to be inserted) and a desired false positive rate, the number of hash functions and the size of the bit vector used can easily be calculated. Figure 9 shows examples of testing a Bloom filter for existing and non-existing elements.

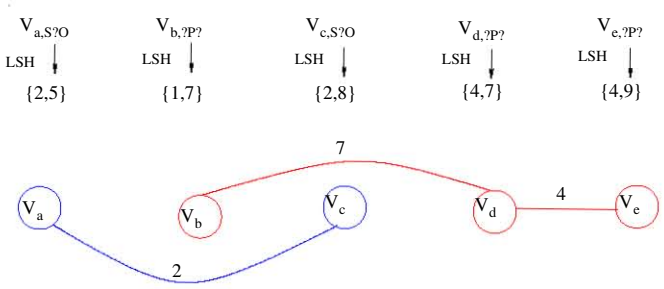


Figure 7. Grouping five PVs into two connected components shown in red and blue ($k = 2, m = 10$)

Algorithm 3 outlines the construction of the PV-Index. The first step is building a graph $\mathbb{G}$ whose vertices represent the individual Pattern Vectors (which represent the input RDF graphs). LSH is applied on each of the seven vectors of the PV 6. If after applying LSH, there is at least one identical hash value for the same pattern r of two different PVs, we draw an edge between the vertices of those two PVs (Lines 2 to 8). This means that two Pattern Vectors are dissimilar with high probability if there is not edge between them. The second step is computing the connected components in the graph $\mathbb{G}$ in linear time using breadth-first search [43]. Individual connected components in $\mathbb{G}$ represent RDF graphs whose Pattern Vectors are similar with high probability. All the RDF graphs in a connected component are treated as one group and the union of their PVs is computed (Line 11) using the Union operation defined previously in this section. Note that the Union operation both summarizes the PVs and preserves the necessary condition in Theorem 3.3. Futhermore, the Union operation can be performed in linear time because the individual vectors in each PV are kept sorted.

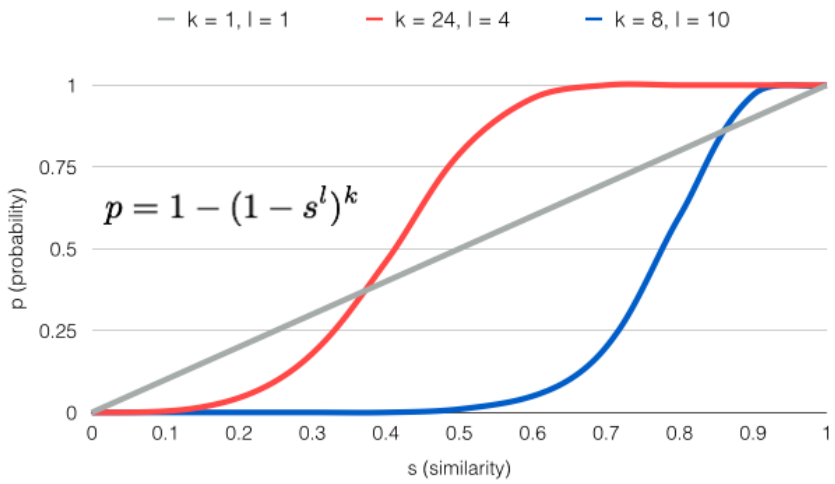
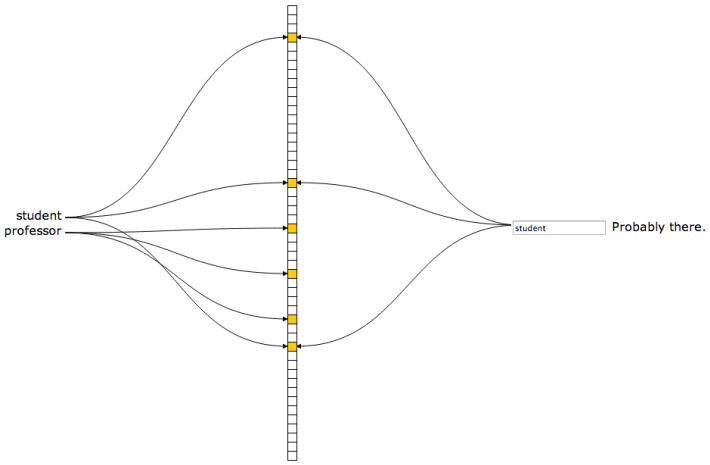
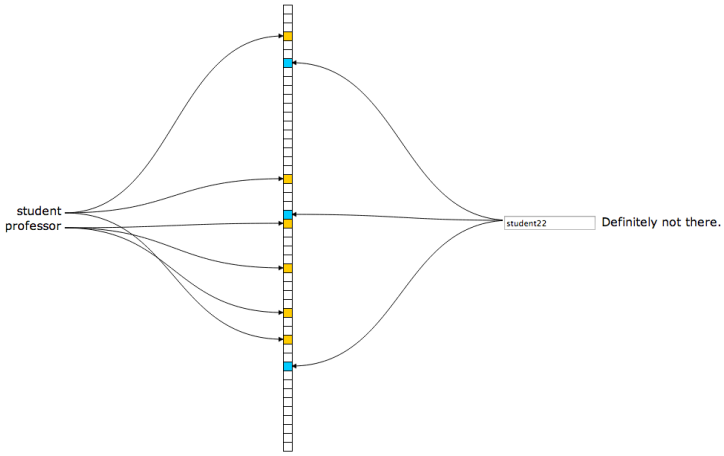


Figure 8. LSH: probability vs similarity

We use a combination of Bloom filters (BFs) and Counting Bloom filters (CBFs) to represent the union of PVs in each connected component. For the SPO canonical pattern, we use a Bloom filter because that vector is a set. We use Counting Bloom filters for the vectors of the other six canonical patterns because they are multisets. We set the capacity (the number of elements to be inserted) and the false positive rate ϵ for filter (Lines 12 and 13) where the capacity is equal to the cardinality of the vector. We also store the ids of the graphs belonging to a particular connected component (Line 14). The collection of all BFs and CBFs make up the PV-Index. In addition to constructing the PV-Index, each group of similar graphs is indexed with an RDF application such as RDF-3X, Jena TDB or Virtuoso.



(a) probably there



(b) definitely not there

Figure 9. Bloom filter test operation
(Source: `http://www.jasondavies.com/bloomfilter/`)

Algorithm 3 The PV-Index Construction

Require: a list of PVs; (k, l, m): LSH parameters; ϵ: false positive rate
Ensure: filters of all the groups of similar RDF graphs

Let $\mathbb{G}(\mathbb{V}, \mathbb{E})$ be initialized to an empty graph
for each PV $\overline{V}$ **do**
    Add a new vertex v_i to $\mathbb{V}$
    for each $r \in \mathbb{P}$ **do**
        $\{h_{i1}, ..., h_{ik}\} \leftarrow \mathsf{LSH}_{k,l,m}(V_r)$
        for every $v_j \in \mathbb{V}$ and $i \neq j$ **do**
            if $\exists\ o$ s.t. $1 \leq o \leq k$ and $h_{io} = h_{jo}$ **then**
                Add an edge (v_i, v_j) to $\mathbb{E}$ if not already present
Compute the connected components of $\mathbb{G}$. Let $\{C_1, ..., C_t\}$ denote these components.
for $i = 1$ to t **do**
    Compute the union U_i of all PVs corresponding to the vertices in C_i
    Construct a BF for $U_{i,SPO}$ with false positive rate ϵ given the capacity $|U_{i,SPO}|$
    Construct a CBF for each of the remaining vectors of U_i with false positive rate ϵ given the capacity $|U_{i,*}|$
    Store the ids of graphs belonging to C_i
return

3.3 Query Processing

We take a *decrease & conquer* approach to processing SPARQL queries in RIQ. First, using the PV-Index, we identify candidate groups of RDF graphs which with high probability might contain a match for the BGPs in the query. Next, we execute the optimized (and potentially re-written) SPARQL queries only on those candidates and skip a large number of groups without false dismissals.

We start by parsing the given SPARQL query's `GRAPH` block based on the supported SPARQL grammar (Appendix B). We generate a parse tree which we call the BGP Tree. The BGP Tree serves as the query execution plan for processing the individual BGPs. Figure 10 shows an example of such a parse tree. Algorithm 4 shows the pseudo-code for the evaluation of the BGP tree. We maintain the status of the evaluation of each connected component (group of RDF graphs) of the PV-Index by keeping track of it in a variable called $eval[n]$ where n is the id of the node. We initialize *eval* for all nodes to `FALSE`. In depth-first order, we invoke Algorithm 4 on each connected component and recursively process the individual nodes in a connected component. We skip children of `GroupGraphPatternSub` which evaluate to `FALSE` (Line 4). This is a part of the query optimization technique and is possible because it is certain that the RDF graphs in that particular connected component will not produce for the subexpressions rooted at that node. However, subexpressions rooted at `GroupOrUnionGraphPattern` will evaluate to `TRUE` if at least one of their children evaluate to `TRUE` (Line 7).

We test the necessary condition defined in Theorem 3.3 upon reaching a leaf node which is a BGP (a `TriplesBlock` and call Algorithm 5. In this algorithm, we

have to construct query PVs on the fly (Line 2) because the capacity of the data PVs varies and comparing BFs and CBFs (Line 3) with different capacities in order to establish if one is a subset of the other is meaningless. Finally, back in Algorithm 4, if an `OptionalGraphPattern` node evaluates to `TRUE`, we return `TRUE` because of the semantics of `OPTIONAL` in SPARQL. A connected component (a group of RDF graphs) is considered a candidate if the root of the tree ($eval[n]$) evaluates to `TRUE`. Once a group of RDF graphs is identified as a candidate, we are able to generate an optimized SPARQL query by traversing the BGP Tree and pruning parts of it based on Algorithm 6. For certain predicates such as `FILTER`, we include all of the subpredicates they include (*e.g.* `NOT EXISTS`). In order to be able to combine the results from multiple candidates successfully, we have to project all the variables from the original query, even if they no longer appear in the optimized query due to pruning of the BGP which contained those variables. Figure 11 shows an example of a optimized SPARQL query generated after running the pruning algorithm. Note that both the `OPTIONAL` and the `UNION` blocks are absent. Excluding those BGPs during query execution can significantly speed up query time as shown in Chapter 5. Finally, we execute the optimized SPARQL query on the candidates graphs by using an application such as Jena TDB or Virtuoso. The results of the output from all the candidates are combined in a trivial manner as no graphs span between candidate groups.

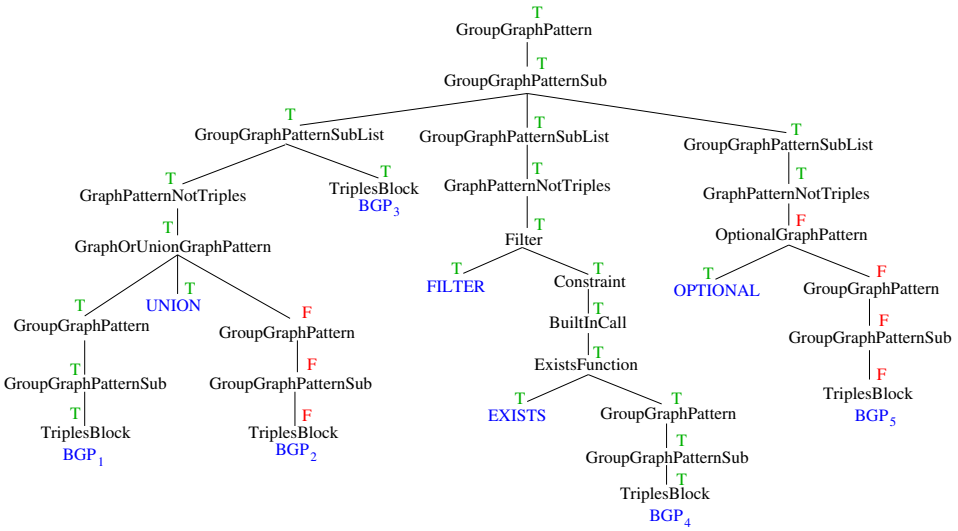


Figure 10. An example of a BGP Tree

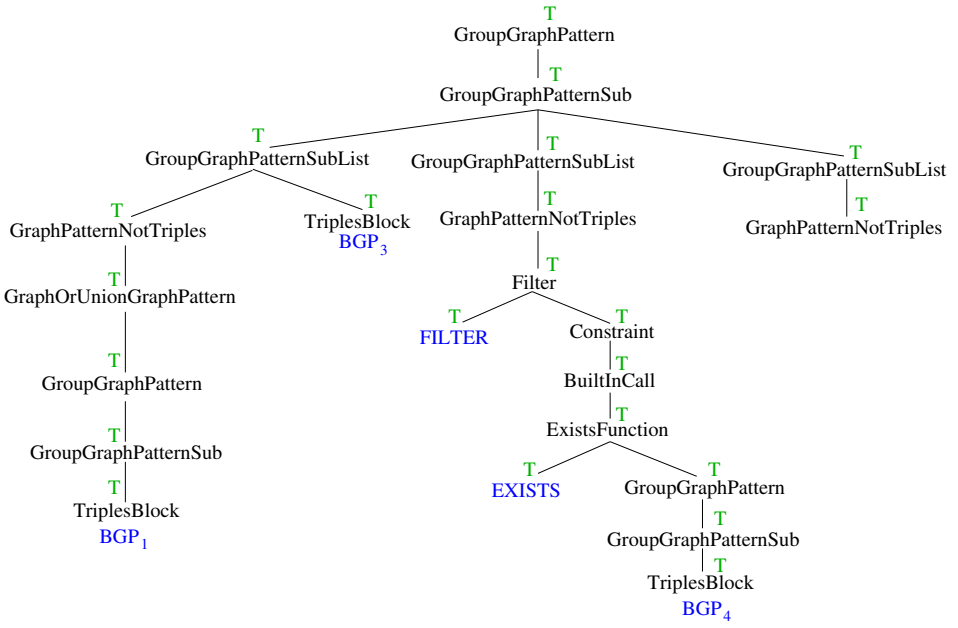


Figure 11. The BGP Tree after pruning

Algorithm 4 EvalBGPTree(node n, conn. component j)

Let $c_1, ..., c_\tau$ denote the child nodes of n (left-to-right)
for $i = 1$ to τ **do**
    $eval[c_i] \leftarrow$ EvalBGPTree(c_i, j)
    if n is `GroupGraphPattern` & $eval[c_i]$ = `FALSE` **then**
        $eval[n] \leftarrow$ `FALSE`
        return `FALSE` {//skip rest of the nodes}
if n is `GroupOrUnionGraphPattern` **then**
    $eval[n] \leftarrow \bigvee_{i=1}^{\tau} eval[c_i]$
else if n is `EXISTS` **then**
    $eval[n] \leftarrow eval[c_1]$
else if n is `NOT EXISTS` **then**
    $eval[n] \leftarrow$ `TRUE`
else if n is `Predicate` **then**
    $eval[n] \leftarrow$ `TRUE` {//skip processing predicates}
else if n is `BGP` **then**
    Let q denote the basic graph pattern
    $eval[n] \leftarrow$ IsMatch(q, j)
else
    $eval[n] \leftarrow eval[c_\tau]$
if n is `OptionalGraphPattern` **then**
    return `TRUE`
return $eval[n]$

Algorithm 5 IsMatch(BGP q, conn. component j)

For connected component j, let $\mathbb{F}_{j,r}$ denote the BF or CBF constructed for pattern r
Construct $\mathbb{F}_{q,r}$ with the same capacity and false positive rate as $\mathbb{F}_{U_j,r}$
if (1) for each bit in $\mathbb{F}_{q,SPO}$ set to 1, the corresponding bit in $\mathbb{F}_{U_j,SPO}$ is 1, and (2) for each of the remaining patterns, given a non-zero counter in $\mathbb{F}_{q,r}$, the corresponding counter in $\mathbb{F}_{U_j,r}$ is greater than or equal to it **then**
    return `TRUE`, otherwise **return** `FALSE`

Algorithm 6 PruneBGPTree(node n)

```
1: Let c_1, ..., c_τ denote the child nodes of n (left-to-right)
2: if eval[n] = FALSE then
3:    if n's parent is NOT EXISTS then
4:       return TRUE
5:    else if n is OptionalGraphPattern then
6:       return FALSE
7:    else if n is GroupGraphPattern & left-sibling is UNION then
8:       Prune away the subtree rooted at the left-sibling of n
9:    Prune away the subtree rooted at n from the BGP Tree
10: else
11:    for i = 1 to τ do
12:       status ← PruneBGPTree(i)
13:       if status = FALSE then
14:          Prune away the subtree rooted at i from the BGP Tree
15: return TRUE
```

CHAPTER 4

IMPLEMENTATION OF RIQ

4.1 System Architecture

RIQ consists of three distinct engines: the Indexing Engine, the Filtering Engine, and the Execution Engine (Figure 12).

The Indexing Engine is responsible for four tasks. The first task is the construction of Pattern Vectors from the input RDF graphs. Algorithm 1 in Chapter 3

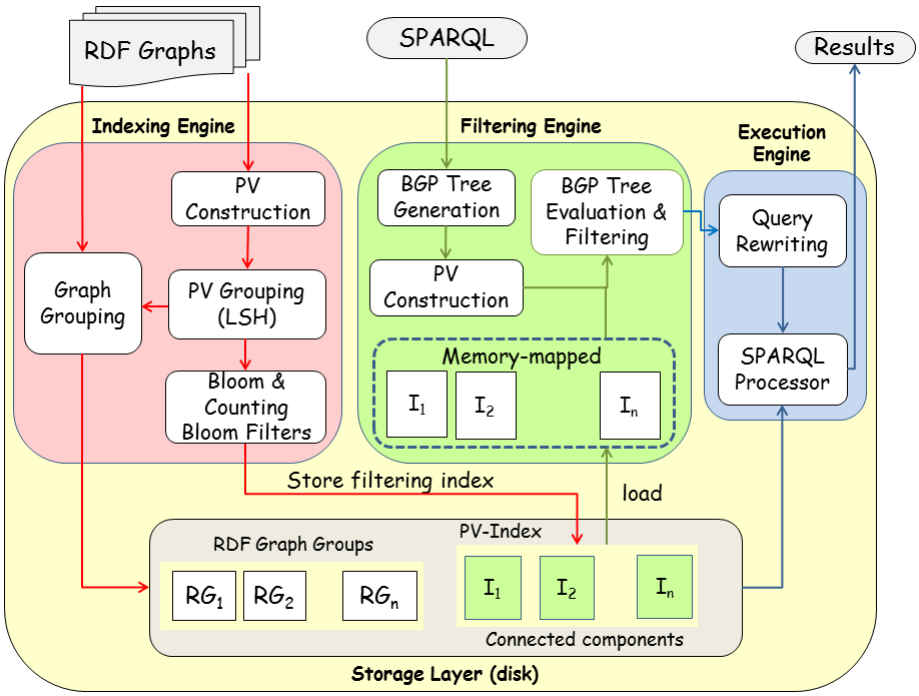


Figure 12. Architecture of RIQ

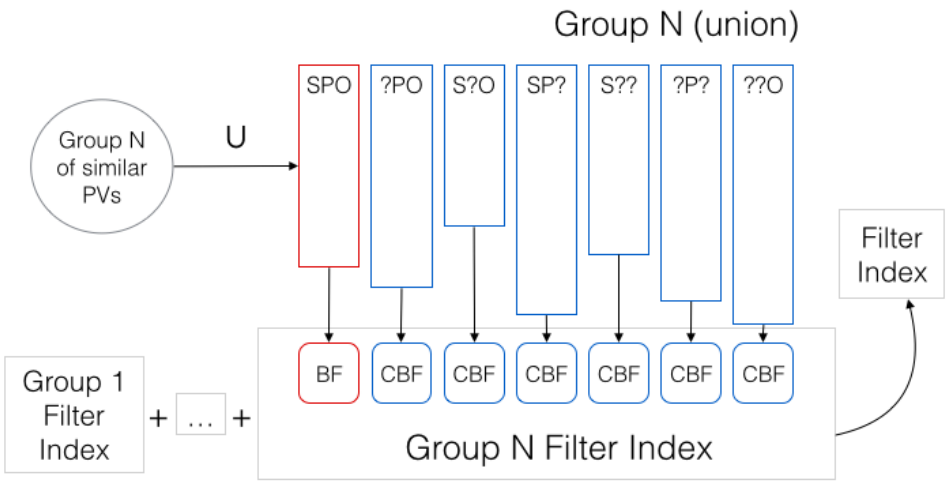


Figure 13. Filtering Index generation in RIQ

describes the PV construction. Having transformed all RDF graphs into sets and multisets of non-negative integers (the Pattern Vectors), the Indexing Engine employs Locality Sensitive Hashing (LSH) [35] to perform similarity-based grouping as described in Algorithm 3 of Chapter 3. As the individual Pattern Vectors are grouped, their corresponding RDF graphs are written to disk into separate files (one per group) where each file/group contains many RDF graphs. In addition, Bloom filters and Counting Bloom filters are generated for the grouped PVs. The BFs and CBFs are stored to disk. The collection of all BFs and CBFs for all the unions of grouped PVs comprise the filtering index called PV-Index. Figure 13 illustrates the steps and transformations.

The Filtering Engine in RIQ performs several tasks at query time. Those tasks are a part of the filtering phase of query processing where candidates of RDF graphs

which may match the query are identified. First, the Filtering Engine generates a parse tree which we call a BGP Tree out of the SPARQL query which serves as the execution plan for processing the individual BGPs. Next, the Pattern Vector for each BGP is constructed as described in Algorithm 2. This PV along with the memory-mapped PV-Index of the RDF graphs serve as the input for the BGP Tree Evaluation & Filtering task which is described in detail in Algorithm 4. Figure 14 shows the steps in testing if a group is a candidate for a query PV by using the subset operation on the corresponding vectors within the PV. The subset testing is performed using the Bloom filters and Counting Bloom filters by comparing the individual bits and counters of the data and query filters. Note that the query BFs and CBFs are constructed at query time in order for their capacity and error rate to correspond to the data BF/CBF which is being tested. If those parameters are not the same, comparing the bits and counters would be meaningless as the number of hash functions used for inserting would be different. For Bloom filters, the corresponding bits in the query BF and the data BF are checked such that $BF_q[i] == BF_u[i]$. For Counting Bloom filters, the counters in the query CBF are compared to the corresponding counters in the data CBF such that $CBF_q[i] \leq CBF_u[i]$.

Finally, the Execution Engine performs the final two tasks. First, it does query rewriting by pruning the BGP Tree as described in Algorithm 6. The rewritten query can be much more efficient to process because BGPs for which there will be no matches are excluded. The rewritten queries are handed to the SPARQL Processor (an application such as Jena TDB or Virtuoso) which runs each of the rewritten queries on the corresponding candidate group indexes and outputs the results for

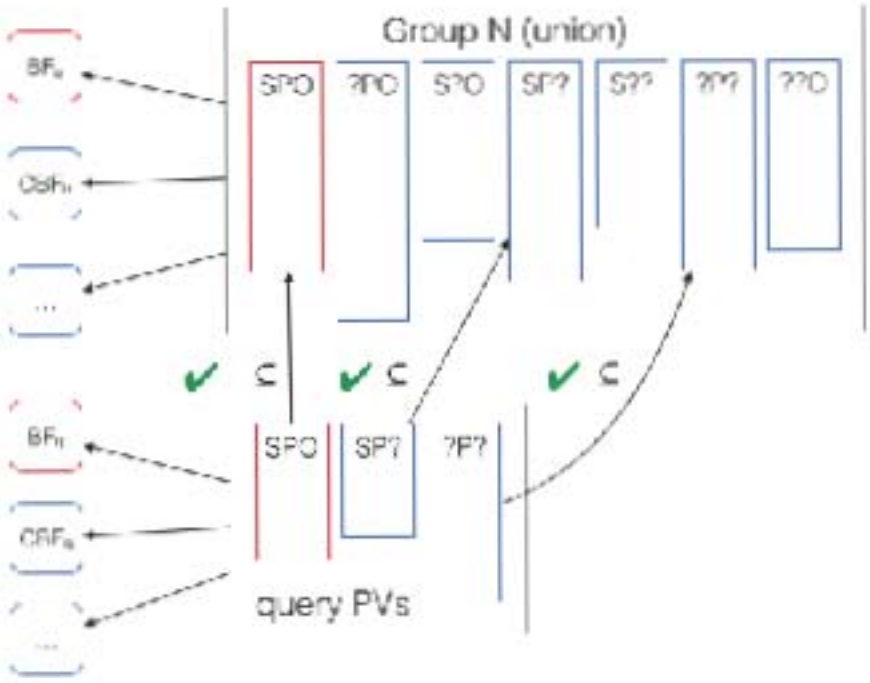


Figure 14. Query execution in RIQ

each group. Since RDF graphs do not span groups, merging of the results is trivial: they are concatenated. Note that in order for the merging to be successful, all the variables from the original SPARQL query need to be projected in the re-written, optimized queries (even if those variables are not used in the BGPs of the candidate query because they were pruned away).

4.2 Implementation

RIQ was implemented in C++ and was running as a single-threaded application. In addition to relying heavily on the C++ STL, we also used a number of external libraries. For RDF parsing of triples and quadruples, we used the highly efficient C Raptor library [20]. For SPARQL query parsing and rewriting, we used the closely-related Rasqal [21] library. We implemented our own version of LSH [38] which we had already used successfully in some of our previous papers [54, 55, 52]. For Bloom Filters and Counting Bloom Filters, we made some minor modifications to the popular Bit.ly Dablooms library [23] which allowed us to control the precision and capacity of our Bloom Filters. Finally, during indexing, we used Google's sparse_hash_map [3] for the efficient storing of very large in-memory hash maps of LSH buckets when building the filtering index.

We integrated three different SPARQL processors in RIQ. For RDF-3X, we used the package provided by the authors and called it using Python scripts. For Jena TDB and Virtuoso, we wrote Java applications which called their corresponding APIs.

4.3 Limitations

RIQ was designed to excel at processing queries with large, complex BGPs. As we will show in the Evaluation chapter, RIQ is able to process those queries significantly faster than both RDF-3X and Jena TDB. For queries with small, mostly star-shaped BGPs, RIQ has a competitive performance.

At the center of RIQ's design is a successful grouping of the RDF graphs in the dataset based on their similarity using LSH where the number of groups of highly-similar RDF graphs is much smaller than the total number of graphs. If the input RDF data is very homogenous, in the worst case, RIQ will create one giant group of all the graphs. On the other hand, if the dataset contains RDF graphs which are highly dissimilar, in the worst case, RIQ will create a group for every single RDF graph. In both cases, query processing times are negatively impacted. In the first case, the one large group of graphs is equivalent to indexing the dataset with Jena TDB without using RIQ and using RIQ becomes unnecessary. The query time will be larger because it will include RIQ's filtering time for identifying the single group. In addition, the creation of the PV-Index will significantly increase the total indexing time without providing any benefit at query time.

In the second case, where every single graph is represented by a separate group (*i.e.* the number of graphs is equal to the number of groups), depending on the query, RIQ is likely to identify a large number of candidates during filtering which will slow down the SPARQL processor (*e.g.* Jena TDB) because it will have to query a large number of small indices. The I/O is likely to increase significantly. The difference between cold and warm query times will likely decrease (the warm times

will get slower) due to the high number of page faults. In addition, both the size of the filtering index (PV-Index) and the total size of the indexed RDF data will be the largest possible because it will not be possible to take advantage of compressing graphs with common subjects, predicates or objects.

Related to the above worst case scenarios is the difficulty of figuring out the "perfect" LSH k and l parameters (the number of hash functions used for grouping) such that the worst cases are avoided. This limitation is inherent to LSH as it is highly-dependent on the input data. Despite this limitation, LSH has been successfully deployed over the years in a large number of applications in many domains in both research and industry [60]. We explored an alternative method for grouping of RDF graphs in RIQ. Instead of the probabilistic, estimate approach which LSH takes, we implemented a prototype of RIQ which calculated the exact Jaccard index of the PVs representing the RDF graphs in order to decide how to group them. Unfortunately, this calculation made indexing prohibitively expensive (the indexing time grew from a few hours to a few days) without providing a significant improvement at query time. A comparison of the query times for RIQ using LSH and RIQ using Jaccard for grouping can be found in Tables 12 and 14 in the Evaluation chapter.

Another limitation which is related to the size of the groups created by LSH has to do with keeping the size of the filtering index small to minimize the I/O during filtering. The size of the PV-Index is a function of the size of the individual Bloom filters and Counting Bloom filters which represent the groups of similar RDF graphs. If we allow for the BF and CBF capacities to be equivalent to the size of the groups they represent, we risk having a very large PV-Index. On the other hand, if we set

a maximum capacity of the BFs and CBFs, we risk having too many false positives as the counters in the CBFs will start overflowing and all the bits in the BFs will be set. Two ways in which we try to minimize the I/O during filtering is first by implementing the PV-Index as a collection of memory-mapped BFs and CBFs. In addition, we choose the LSH k and l parameters, and the maximum capacity and false positive rate of the BFs and CBFs such that the size of the PV-Index is smaller than the available RAM.

While RIQ is very fast at processing queries with large, complex BGPs, it struggles with low selectivity queries which contain a match in most or all the RDF graphs. The evaluation section shows the results for two such queries (L5 and L7) which output all the graduated and undergraduate students in the LUBM dataset. The number of results for those queries is 25 and 79 million respectively. Because those queries contain a match in every single RDF graph, it is impossible for RIQ to group the graphs in such a way that the number of candidate groups is not equal to the total number of groups. This forces the SPARQL processor to query every single group in the refinement phase. One way to minimize the negative impact of such queries is to minimize the total number of groups.

CHAPTER 5

EVALUATION

In this chapter, we present the performance evaluation of RIQ. We have compared RIQ with the latest versions of three of the state of the art applications for indexing and query processing of RDF data: RDF-3X 0.3.8 [46], Apache Jena TDB 2.11.1 [18], and OpenLink Software Virtuoso OpenSource 7.10 [56]. Unlike RDF-3X, both Jena TDB and Virtuoso support RDF quads. The machine on which we conducted all the experiments was running a 64-bit Ubuntu 13.10 with Linux kernel 3.11.0 and had a 4-core Intel Xeon 2.4GHz CPU and 16GB RAM.

For Jena TDB, the default statistics-based TDB optimization was used because the other settings are intended for exploring the optimizer strategy. We set the Java max heap size to 8GB (half of the available RAM). For Virtuoso, we set a number of memory-related parameters recommended by the OpenLink documentation for the amount of RAM on the machine where Virtuoso is run. In particular, we set the NumberOfBuffers to 1360000 and MaxDirtyBuffers to 1000000. Futhermore, we set the MaxCheckpointRemap to 1000000. The recommended 1/4 of the NumberOfBuffers setting caused too many re-mappings and the Virtuoso logs suggested increasing the parameter. Finally, we did not set the ShortenLongURIs parameter as we did not want to give Virtuoso an unfair advantage in the comparison with the rest of the approaches.

Note that the comparison with Virtuoso is more limited due to problems with

indexing the full BTC triples dataset which was used for running a couple of the sets of query types. After trying to index the BTC triples dataset for 6 days with Virtuoso, we gave up. However, we were able to index the BTC quad dataset for testing the multi-BGP queries.

5.1 Datasets

The evaluation was done using two datasets widely used in the Semantic Web, one synthetic and the other one real. The synthetic dataset was the Lehigh University Benchmark (LUBM) [7]. We generated 200,004 files (and 10,000 universities) which we considered as separated RDF graphs. The dataset contained of 1.38 billion triples. There were 18 unique predicates, 216,971,360 subjects, and 161,394,242 objects. There were less than 100 RDF types.

The second, real, dataset was BTC-2012 [11]. It contained 1.36 billion quads. The Billion Triple Challenge is comprised of 5 different sources of real data from different web resources: Datahub, DBpedia, Freebase, Rest, and Timbl. The total number of RDF graphs was considerably higher than LUBM: 9.59 million. Furhtermore, BTC contained a much larger number of unique predicates: 57,000. There were 183,000,000 subjects and 192,000,000 objects. In addition, BTC-2012 contained 156,000,000 literals and 296,000 different RDF types.

5.2 Queries

As explained in Chapter 2, RIQ is geared towards queries with large, complex BGPs. There were three main types of queries we evaluated: queries containing large, complex BGPs, queries with small BGPs, and finally, queries with multiple BGPs.

Table 2 shows more information about each set of queries, including the number of results and triple patterns. Appendix A lists the full text of all the SPARQL queries.

The first set of queries contained many triple patterns (at least 11 and at most 22) and most of the queries contained undirected cycles. There were 3 LUBM (L1-L3) and 2 BTC large, complex BGP queries. Appendix A contains a visualization of these queries.

The second set were queries similar to the ones used in other research publications: small, mostly star-shaped and some short path/chain queries. The 9 small LUBM queries (L4-L12) are variations of queries from the 14 standard LUBM benchmark queries. Since RDF-3X does not support inferences, we used instances of the inference types from the original queries. For BTC, we used 5 small queries (B3-B7). Both the large and the small queries contained single BGPs.

For the last type, multi-BGP queries, we modified queries from the DBpedia SPARQL benchmark [44], one of the only real (non-synthetic) SPARQL benchmarks. The DBPSB is based on real queries extracted from the `dbpedia.org` logs. The queries contain a wide range of SPARQL grammar and are representative of real-world SPARQL usage. The subset of queries we issued used the `UNION` and `OPTIONAL` keywords which enabled us to evaluate our query rewriting optimizations. The queries contained at least 2 and at most 5 different BGPs.

5.3 Indexing

In this section, we report the size and time it took to index the two datasets along with the settings we used. Tables 3 and 4 show all the numbers.

The size of the synthetic, LUBM, dataset was 217GB of RDF triples. The sizes

Table 2. Queries for LUBM and BTC-2012.

Dataset	Query	Type	# of BGPs	# of triple patterns	# of results
LUBM	L1	large	1	18	24
	L2	large	1	11	7,082
	L3	large	1	22	0
	L4	small	1	6	2,462
	L5	small	1	1	25,205,352
	L6	small	1	6	468,047
	L7	small	1	1	79,163,972
	L8	small	1	2	10,798,091
	L9	small	1	6	440,834
	L10	small	1	5	8,341
	L11	small	1	4	172
	L12	small	1	6	0
BTC-2012	B1	large	1	18	6
	B2	large	1	20	5
	B3	small	1	4	47,493
	B4	small	1	6	146,012
	B5	small	1	7	1,460,748
	B6	small	1	5	0
	B7	small	1	5	12,101,709
	B8	multi-bgp	5	8	249,318
	B9	multi-bgp	4	7	149,306
	B10	multi-bgp	7	12	196
	B11	multi-bgp	2	5	525,432

of the RDF-3X and Jena TDB LUBM indices were 77GB and 121GB respectively. The memory-mapped RIQ filter index (PV-Index) was 5.7GB. It consisted of 487 groups of Pattern Vectors of highly similar RDF graphs. It took a little over 4 hours to create the PVs. Table 3 shows the break-down of the PV-Index construction. We identify three phases in the PV-Index construction. The first one is the transformation

from original RDF data to Pattern Vectors reported above. The second one is the grouping of similar PVs by running LSH on the PVs, building a graph with PVs as vertices and connecting PVs which hashed to the same LSH buckets, and finally, calculating the connected components in the group in order to identify the individual groups of graphs. The last phase is the construction of Bloom filters and Counting Bloom filters from the groups of similar Pattern Vectors.

The LSH parameters k and l were set to 30 and 4 respectively. For BF/CBF generation, we set the false positive rate ϵ to 1% and maximum capacity (number of insertions to the filters) to 10^7. Note that for PVs with fewer than the above number of elements, we used the actual size of the PV as the BF/CBF capacity. This combination achieved excellent results in terms of the size of the filter index and the accuracy with which it identified candidate groups. We show the accuracy and efficiency of RIQ in the following sections.

The BTC dataset consisted of 218GB of RDF quadruples. Due to the way it was compiled (through crawling of different Web sources), some of the data did not conform to the RDF standard and required cleaning (reformatting) or exclusion of some URIs and/or literals in order for Jena TDB and RDF-3X to parse the data correctly. We used some of the scripts and techniques used in SPLODGE [32] for cleaning the dataset. The latter's index size was 87GB while the former was 110GB. RIQ's filter index was 6.5GB and consisted of 526 unions of graph PVs. In this case, the LSH parameters k and l were set to 4 and 6 respectively. Because the dataset consisted of 45 times more RDF graphs, the number of LSH hash functions ($k \times l$) could not be very high in order for efficient indexing to occur. The CBF false positive

rate ϵ was 5% and the maximum capacity of the filters was set to 10^6.

Table 3. RIQ's indexing performance: construction time

Dataset	Construction time (in secs)			
	PVs	Grouping	BF/CBFs	Total
LUBM	15,249	22,711	3,402	41,362
BTC-2012	16,700	27,348	2,476	46,523

Table 4. RIQ's indexing performance: settings and size

Dataset	# of unions	False +ve rate (ϵ)	Max. filter capacity	PV-Index size
LUBM	487	1%	10 M	12 GB
BTC-2012	526	5%	1 M	6.5 GB

5.4 Query Processing

We conducted the query processing evaluation by collecting the wall-clock time and I/O stats (major and minor page faults) using `/usr/bin/time`. Two sets of experiments were done: one with cold cache where we dropped the cache using the `/proc/sys/vm/drop_caches` kernel interface and one with warm cache where we ran the same query immediately preceding the warm cache experiment. Each query was run 3 times and the average was taken. In addition, we calculated the geometric mean for the three sets of queries (large, small, multi-BGP) and for all queries from

Table 5. Filtering time for RIQ on LUBM.

Query	**Cold cache** Time taken (in secs)	**Warm cache** Time taken (in secs)
L1	4.03	0.15
L2	6.13	0.15
L3	4.50	0.16
L4	8.87	0.24
L5	4.44	0.10
L6	7.97	0.24
L7	5.89	0.10
L8	4.50	0.16
L9	8.22	0.25
L10	5.28	0.11
L11	5.71	0.11
L12	8.53	0.25

Table 6. Filtering time for RIQ on BTC-2012.

Query	**Cold cache** Time taken (in secs)	**Warm cache** Time taken (in secs)
B1	2.30	0.11
B2	2.10	0.08
B3	2.15	0.10
B4	2.28	0.11
B5	2.14	0.11
B6	6.05	0.14
B7	2.09	0.10
B8	5.15	0.85
B9	5.12	0.78
B10	6.42	0.66
B11	6.21	0.61

a particular dataset.

We report the overall results for LUBM Table 11 and for BTC-2012 in Tables 13 and 15. RIQ was able to process queries with large, complex BGPs (L1-L3) significantly faster than both RDF-3X and Jena TDB for both cold and warm cache settings. For BTC-2012, RIQ processed the large, complex queries B1 and B2 significantly faster than RDF-3X and faster than Jena TDB. These results prove that the *decrease & conquer* approach which RIQ takes is a better choice for processing queries with large, complex BGPs than the popular join-based processing where the individual triple patterns are first matched. All of the large, complex queries contained at least one undirected cycle. Appendix A shows the full text of the SPARQL queries and a visualization of the BGPs in them. RIQ was able to achieve very high precision for those queries. For LUBM queries L1-L3, out of the 487 groups of similar Pattern Vectors, RIQ identified a maximum of 16 candidate groups. For BTC queries B1 and B2, out of the 526 union groups, RIQ identified a maximum of 3 candidate groups. By discarding such a large number of groups during the filtering phase (using the PV-Index), RIQ is able to then run the queries on a much smaller portion of the dataset in the refinement phase using another tool such as Jena TDB or Virtuoso.

We report the performance evaluation of RIQ on queries with small BGPs which contain less than 8 triple patterns: queries L4-L12 and B3-B7. For LUBM, RIQ was faster than both RDF-3X and Jena TDB on four out of the nine queries for both cold and warm cache. For BTC-2012, RIQ was faster than both RDF-3X and Jena TDB in four out of the five queries for cold cache. For warm cache, RDF-3X was the winner for four out of the five queries. While RIQ's advantage was not as strong

for small BGP queries, it was still competitive compared to the other approaches. One reason for the poorer performance on those queries is that they are generally either low-selectivity or contain graph pattern matches within most graphs. For this reason, it was impossible for RIQ to group the graphs in such a way as to minimize the number of candidates for processing during refinement time because most graphs actually contain matches for the query. However, when we compared the geometric mean of the query processing times of the three approaches, RIQ was the clear winner for both LUBM and BTC-2012 across all queries except for one case: the small BGP queries for BTC where RDF-3X won but RIQ was a close second. Tables 9 and 10 show this comparison of the geometric means for the three sets of queries for both LUBM and BTC-2012. Tables 7 and 8 show the fastest approach for processing queries over LUBM and over BTC-2012, respectively.

In the initial stages of developing RIQ, we experimented with using the Jaccard index (*a.k.a* the Jaccard similarity coefficient) [39] for calculating the exact similarity of Pattern Vectors instead of the probabilistic Locality Sensitive Hashing (LSH) [34] approach to similarity. Since it is impossible to exhaustively compare all Pattern Vectors to find the best match, we implemented the Jaccard index for computing the similarity of sets and multisets by using the concept of k highest cardinality sets/multisets. We pick two Pattern Vectors with the k highest cardinality of all the PVs and then we group the rest of the PVs into one of those groups. In cases of same similarity, we assign PVs to smaller groups. We also limit the size of the groups by specifying a fanout size. Tables 12 and 14 show a comparison of the query times for RIQ using the Jaccard index and RIQ using LSH for the grouping of the RDF

graphs. While calculating the Jaccard index was beneficial for some queries (a few of the warm cache LUBM ones), overall, RIQ with LSH was more efficient for query processing. Furthermore, indexing using Jaccard was prohibitively expensive. The grouping phase of indexing was more than 2 times slower for LUBM and more than 3 times slower for BTC-2012 compared to using LSH. For these reasons, we abandoned the Jaccard approach and replaced it with LSH which proved to scale much better for both indexing and query processing.

Finally, we report the results from the query processing stragety of RIQ on SPARQL which contain multiple BGPs and keywords such as `UNION` and `OPTIONAL`. Queries B8 through B11 are multi-BGP queries and are shown in Table 2. They contain at most 5 BGPs and up to 12 triple patterns. One of the 4 queries has high selectivity and the rest are lower selectivity queries. The full text of the queries is listed in Appendix A. Queries B10 and B11 are based on the DBpedia SPARQL Benchmark [44, 45]. RDF-3X does not support such queries and we could not use it in their evaluation. Both Jena TDB and Virtuoso support quads and were able to index the context (graph name) for each triple in BTC-2012.

Table 15 shows the query processing times for all the multi-BGP queries. We turned off the default RIQ query optimization for a baseline comparison. This consisted in doing two things: first, selecting a group as a candidate if any of the BGPs in the query had a match in that group and second, executing the original (non-optimized) query on each candidate. We indicate this way of running RIQ by showing it as 'RIQ (no opt)' in Table 15. It is clearly evident that RIQ was faster with optimization enabled. For the cold cache setting, RIQ was able to beat both

Jena TDB and Virtuoso for all four queries. For the warm cache setting, RIQ beat Jena TDB for two queries and beat Virtuoso for three of the four queries. However, RIQ was first in the geometric mean for cold cache query times and close second after Virtuoso for warm cache query times.

Table 7. Fastest approach for processing queries over LUBM.

Query	**Cold cache**	**Warm cache**
L1	RIQ	RIQ
L2	RIQ	RIQ
L3	RIQ	Jena TDB
L4	RIQ	RIQ
L5	Jena TDB	Jena TDB
L6	RIQ	RIQ
L7	Jena TDB	Jena TDB
L8	RIQ	RDF-3X
L9	RIQ	RIQ
L10	Jena TDB	RIQ
L11	RDF-3X	RDF-3X
L12	RDF-3X	RDF-3X

Table 8. Fastest approach for processing queries over BTC-2012.

Query	**Cold cache**	**Warm cache**
B1	RIQ	RIQ
B2	RIQ	RIQ
B3	RIQ	RDF-3X
B4	RIQ	RDF-3X
B5	RIQ	RIQ
B6	RDF-3X	RDF-3X
B7	RIQ	RDF-3X

Table 9. Geometric Mean of the query processing times for LUBM. Best times are shown in bold within shaded cells.

Query	**Cold cache** Geo. Mean (in secs)			**Warm cache** Geo. Mean (in secs)		
	RIQ	**RDF-3X**	**Jena TDB**	RIQ	**RDF-3X**	**Jena TDB**
L1-L3	**62.72**	2073.92	1595.88	**5.23**	1917.93	74.92
L4-L12	**163.77**	235.94	250.24	**65.17**	116.04	188.37
L1-L12	**128.84**	406.26	397.67	**34.68**	233.97	149.59

Table 10. Geometric Mean of the query processing times for BTC-2012. Best times are shown in bold within shaded cells.

Query	**Cold cache** Geo. Mean (in secs)			**Warm cache** Geo. Mean (in secs)		
	RIQ	**RDF-3X**	**Jena TDB**	RIQ	**RDF-3X**	**Jena TDB**
B1-B2	**6.59**	391.31	14.65	**2**	389.16	12.39
B3-B7	**48.87**	58.74	157.65	6.93	**4.34**	26.7
B1-B7	**27.57**	100.98	79.97	**4.86**	15.68	21.44

Table 11. Query processing times for LUBM. Best times are shown in bold within shaded cells. $X^{\dagger}$ indicates that the query ran for more than X seconds and was terminated.

Query	**Cold cache** Time taken (in secs)			**Warm cache** Time taken (in secs)		
	RIQ	**RDF-3X**	**Jena TDB**	RIQ	**RDF-3X**	**Jena TDB**
L1	**16.38**	68.15	294.84	**2.03**	64.28	3.46
L2	**844.48**	$77,315^{\dagger}$	77,315	**41.43**	$64,637^{\dagger}$	64,637
L3	**17.84**	1692.95	178.3	1.7	1698.01	**1.88**
Geo. Mean (large)	**62.72**	2073.92	1595.88	**5.23**	1917.93	74.92
L4	**211.01**	1967.96	572.61	**133.98**	1898.87	542.92
L5	555.96	1392.1	**317.68**	499.2	657.55	**175.48**
L6	**437.76**	1156.04	1121.31	**374.13**	849.13	1243.61
L7	844.51	1455.9	**683.13**	826.2	1298.34	**672.83**
L8	**477.83**	1212.67	1468.28	408.79	**68.42**	1817.32
L9	**467.01**	1262	1140.09	**400.59**	1236.12	1158.5
L10	11.57	9.03	**5.84**	**0.56**	6.38	2.45
L11	11.01	**1.7**	4.26	0.32	**0.26**	1.12
L12	68.77	**20.92**	663.11	34.95	**19.75**	648.26
Geo. Mean (small)	**163.77**	235.94	250.24	**65.17**	116.04	188.37
Geo. Mean (all)	**128.84**	406.26	397.67	**34.68**	233.97	149.59

Table 12. Query times of LSH and Jaccard using RIQ for LUBM. Best times are shown in bold within shaded cells.

Query	**Cold cache** Time taken (in secs)		**Warm cache** Time taken (in secs)	
	RIQ **(Jaccard)**	RIQ **(LSH)**	RIQ **(Jaccard)**	RIQ **(LSH)**
L1	28.09	**16.38**	2.8	**2.03**
L2	**300.08**	844.48	**49.03**	41.43
L3	36.47	**17.84**	4.14	**1.7**
L4	229.95	**211.01**	**27.46**	133.98
L5	576.96	**555.96**	567.2	**499.2**
L6	506.93	**437.76**	489.36	**374.13**
L7	892.7	**844.51**	871.12	**826.2**
L8	507.43	**477.83**	497.69	**408.79**
L9	538.99	**467.01**	519.22	**400.59**
L10	18.72	**11.57**	**0.51**	0.56
L11	12.19	**11.01**	0.41	**0.32**
L12	103.14	**68.77**	**26.76**	34.95
Geo. Mean	148.9	**128.84**	**36.43**	34.68

Table 13. Query processing times for BTC-2012. Best times are shown in bold within shaded cells.

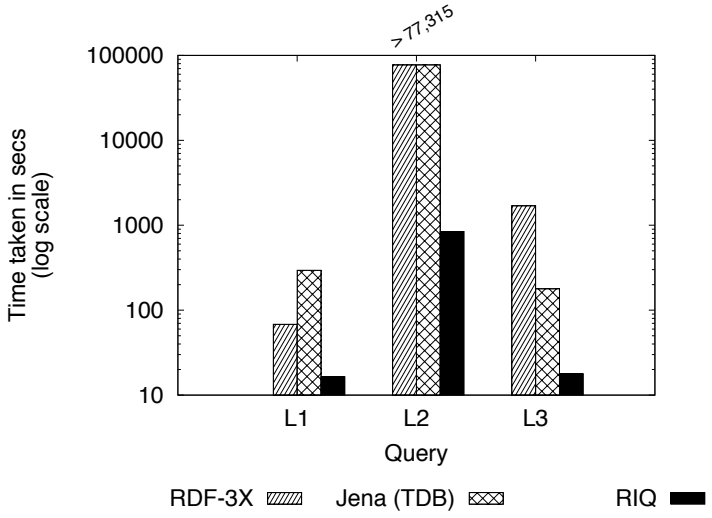
Query	**Cold cache** Time taken (in secs)			**Warm cache** Time taken (in secs)		
	RIQ	**RDF-3X**	**Jena TDB**	RIQ	**RDF-3X**	**Jena TDB**
B1	**5.99**	419.02	12.5	**1.37**	414.53	10.09
B2	**7.25**	365.44	17.18	**2.92**	365.34	15.21
Geo. Mean (large)	**6.59**	391.31	14.65	**2**	389.16	12.39
B3	**37.46**	194.93	357.34	1.82	**0.8**	13.42
B4	**36.12**	161.72	290.34	3.6	**2.35**	17.07
B5	**59.55**	188.78	319.41	**25.55**	28.43	46.16
B6	16.13	**0.41**	11.98	0.52	**0.16**	10.3
B7	**214.37**	286.64	245.3	183.61	**180.22**	124.47
Geo. Mean (small)	**48.87**	58.74	157.65	6.93	**4.34**	26.7
Geo. Mean (all)	**27.57**	100.98	79.97	**4.86**	15.68	21.44

Table 14. Query times of LSH and Jaccard using RIQ for BTC-2012. Best times are shown in bold within shaded cells.

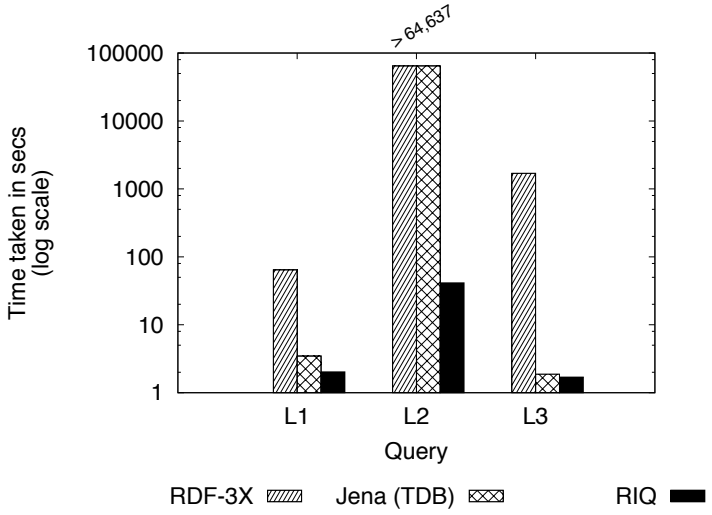
Query	**Cold cache** Time taken (in secs)		**Warm cache** Time taken (in secs)	
	RIQ **(Jaccard)**	RIQ **(LSH)**	RIQ **(Jaccard)**	RIQ **(LSH)**
B1	8.81	**5.99**	1.19	**1.37**
B2	14.56	**7.25**	6.52	**2.92**
B3	41.01	**37.46**	1.83	**1.82**
B4	42.17	**36.12**	**3.59**	3.6
B5	70.15	**59.55**	32.38	**25.55**
B6	20.39	**16.13**	0.64	**0.52**
B7	221.86	**214.37**	184.86	**183.61**
Geo. Mean	35.45	**27.57**	5.7	**4.86**

Table 15. Query processing times for multi-BGP BTC-2012 queries. Best times are shown in bold within shaded cells.

Query	**Cold cache** Time taken (in secs)				**Warm cache** Time taken (in secs)			
	RIQ	RIQ **(no re-writing)**	**Jena TDB**	**Virt-uoso**	RIQ	RIQ **(no re-writing)**	**Jena TDB**	**Virt-uoso**
B8	**116.74**	128.06	663.31	165.52	62.61	76.44	**38.77**	88.36
B9	**110.7**	122.76	648.93	142.89	57.82	74.73	**33.36**	60.42
B10	**16.29**	495.81	3564.44	39.18	6.79	355.07	369.81	**0.16**
B11	**158.18**	163.05	2052.62	237.58	**76.68**	90.09	2102.06	120.28
Geo. Mean	**75.96**	201.06	1331.83	121.81	37.05	110.33	178.07	**17.9**

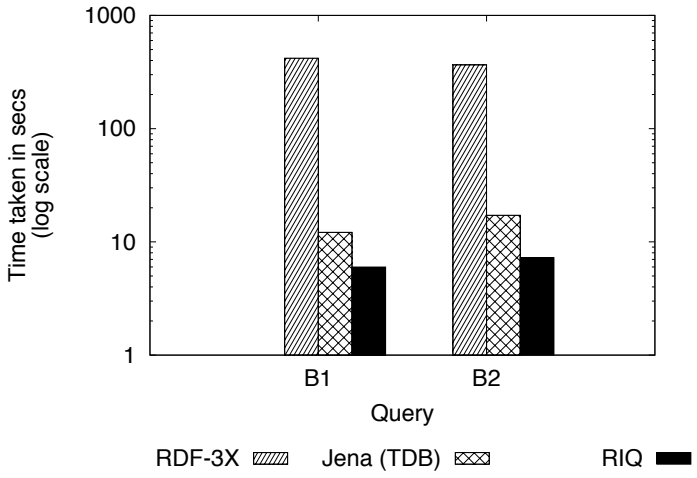


(a) cold cache

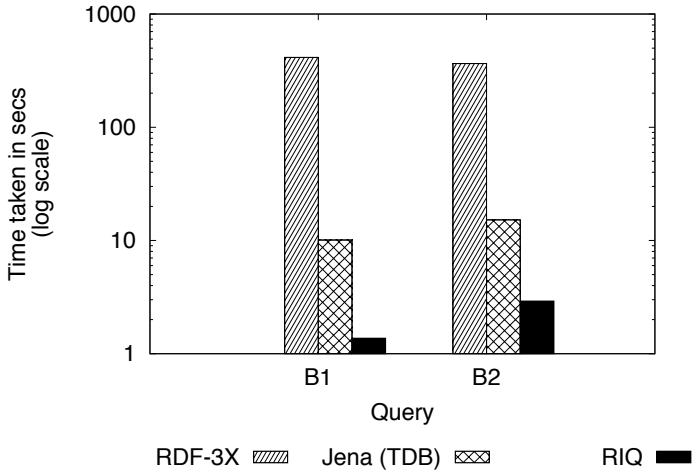


(b) warm cache

Figure 15. Query processing times for LUBM, large queries

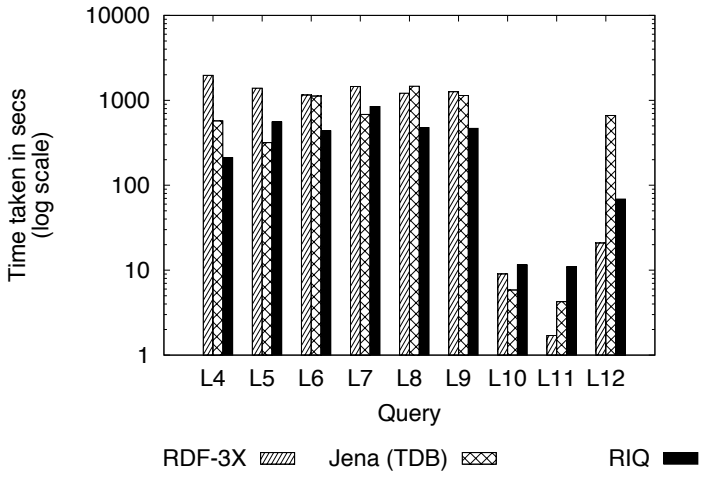


(a) cold cache

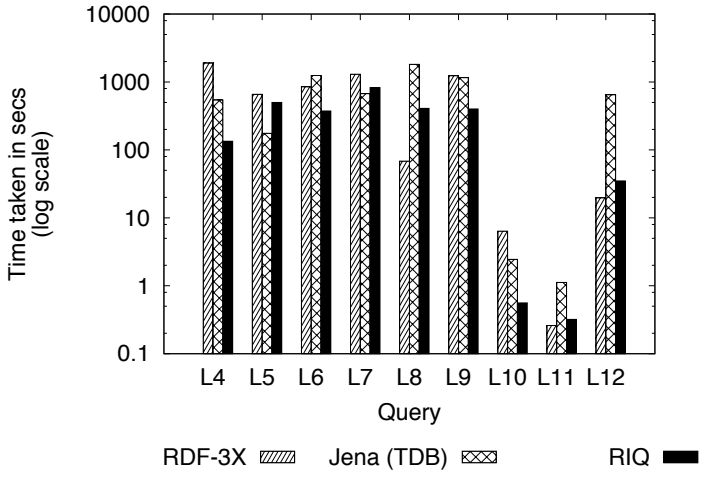


(b) warm cache

Figure 16. Query processing times for BTC-2012, large queries

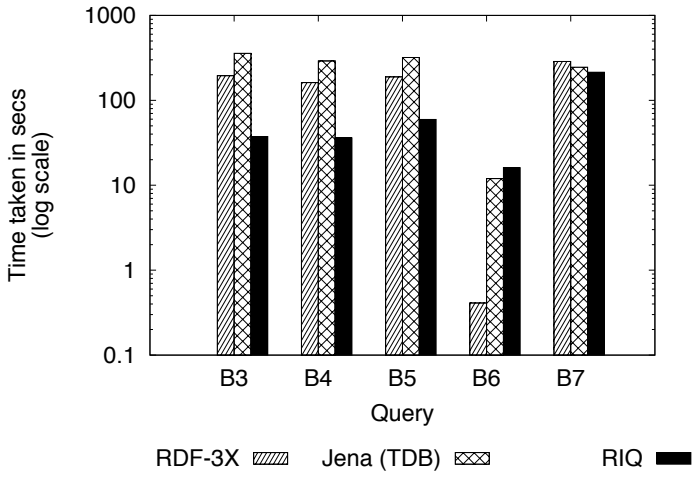


(a) cold cache

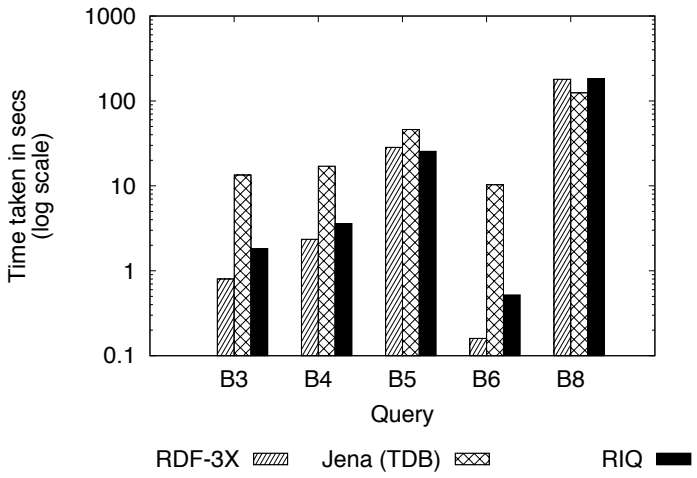


(b) warm cache

Figure 17. Query processing times for LUBM, small queries

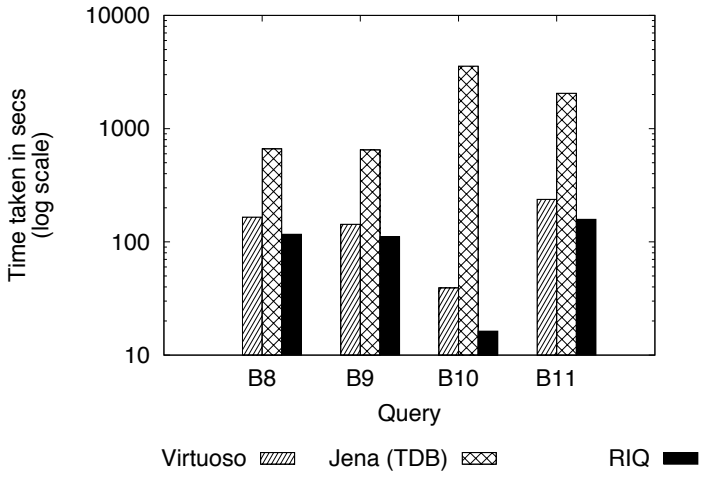


(a) cold cache

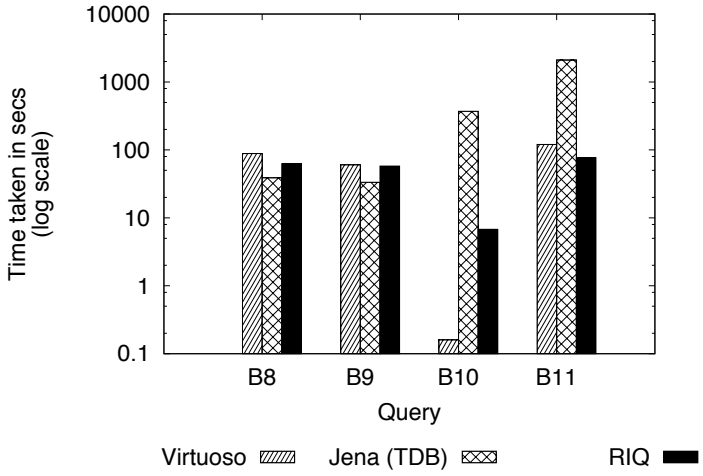


(b) warm cache

Figure 18. Query processing times for BTC-2012, small queries



(a) cold cache



(b) warm cache

Figure 19. Query processing times for BTC-2012, multi-BGP queries

CHAPTER 6

CONCLUSION AND FUTURE WORK

We presented a new application called RIQ for indexing large RDF datasets containing quadruples. RIQ employs a *decrease & conquer* approach for efficiently processing SPARQL queries. RIQ indexes RDF graphs by first transforming them to Pattern Vectors and then grouping the PVs using Locality Sensitive Hashing. A filtering index called PV-Index is built from the groups of similar PVs by constructing Bloom filters and Counting Bloom filters for each PV. At query time, RIQ uses the filtering index to identify candidate groups of RDF graphs which may contain a match for the query. RIQ is able to discard a large number of groups as non-candidates without false dismissals. A query execution plan is generated from the original SPARQL query in the form of a parse tree. The parse tree is evaluated and parts of the tree are pruned based on the results from the filtering index. New, optimized SPARQL queries are generated for each identified candidate group. RIQ executes these optimized SPARQL queries using a conventional SPARQL processor such as Jena TDB or Virtuoso to obtain the final query results. In the comprehensive performance evaluation, we demonstrated that RIQ is able to efficiently process queries with large, complex BGPs on large RDF datasets containing billions of quadruples. RIQ was able to significantly outperform tools such as RDF-3X and Jena TDB for large, complex queries and queries with multiple BGPs and it achieved competitive performance for queries with small BGPs.

For future work, we plan to expand the supported SPARQL grammar to include the full SPARQL specification. In addition to providing full support for `FILTER` expressions, we would also like to introduce support for queries with multiple `GRAPH` patterns (e.g. for matching BGPs in different graphs and combining those `GRAPH` patterns with keywords such as `UNION`). The query planning algorithm will need to be revised in order to support those types of queries. Furthermore, we would like to deploy and evaluate RIQ in a distributed setup (e.g. as a Linked Data application) for evaluating federated SPARQL queries using the `SERVICE` keyword. It will be interesting to compare RIQ and Jena TDB running as Linked Data applications. Finally, we would like to explore support for updated and new RDF data without having to re-index the full dataset. Indexing of new data is significantly easier as it will only require creating Pattern Vectors for the new RDF graphs and adding them to the appropriate existing groups (if similar to any) or creating new groups. Updating the PV-Index will be trivial as both Bloom filters and Counting Bloom filters support adding new items. However, support for updated RDF graphs without having to re-index the full dataset is a non-trivial task as BFs and CBFs do not support deleting items without introducing false negatives.

APPENDIX A

QUERIES

A.1 LUBM Queries

Large BGP queries L1, L2, and L3 are shown below. Figures 20, 21, and 22 show the visual representation of each query's BGP. The common prefixes are listed below.

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

L1: A Full Professor who is an advisor to a graduate and an undergraduate student in a specified university's department and who is a publication co-author with the graduate student:

```
SELECT ?p ?c ?e ?ph ?res ?uguni ?msuni ?phduni ?s1n ?s2n ?s1 ?s2 ?pub WHERE {
  GRAPH ?g {
    ?s1 ub:advisor ? .
    ?s1 ub:name ?s1n .
    ?s1 rdf:type ub:UndergraduateStudent .
    ?s2 ub:advisor ?p .
    ?s2 ub:name ?s2n .
    ?s2 rdf:type ub:GraduateStudent .
    ?p rdf:type ub:FullProfessor .
    ?p ub:name "FullProfessor7" .
    ?p ub:teacherOf ?c .
    ?p ub:undergraduateDegreeFrom ?uguni .
    ?p ub:mastersDegreeFrom ?msuni .
    ?p ub:doctoralDegreeFrom ?phduni .
```

```
    ?p ub:worksFor <http://www.Department17.University1001.edu> .
    ?p ub:emailAddress ?e .
    ?p ub:telephone ?ph .
    ?p ub:researchInterest ?res .
    ?pub ub:publicationAuthor ?p .
    ?pub ub:publicationAuthor ?s2 .
  }
}
```

L2: Two different graduate students who got their undergraduate degrees from the same university, attend the same specified graduate university and are publication co-authors:

```
SELECT ?s1 ?s2 ?pub ?uguni ?dept WHERE {
  GRAPH ?g {
    ?s1 rdf:type ub:GraduateStudent .
    ?s1 ub:undergraduateDegreeFrom ?uguni .
    ?s1 ub:memberOf ?dept .
    ?s2 rdf:type ub:GraduateStudent .
    ?s2 ub:undergraduateDegreeFrom ?uguni .
    ?dept rdf:type ub:Department .
    ?dept ub:subOrganizationOf <http://www.University1167.edu> .
    ?uguni rdf:type ub:University .
    ?pub rdf:type ub:Publication .
```

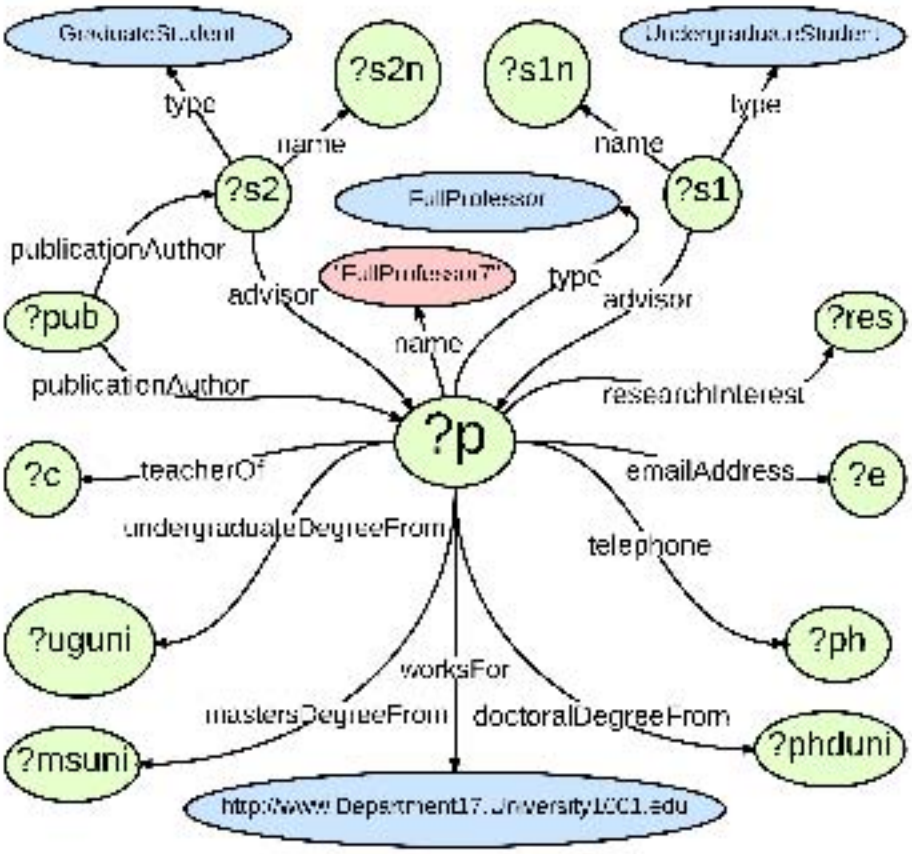


Figure 20. Visual representation of LUBM query L1 with a large, complex BGP

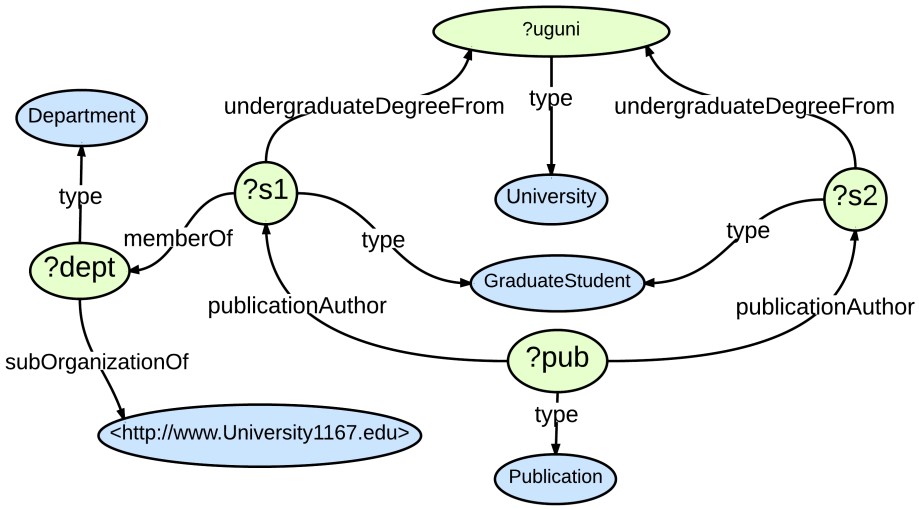


Figure 21. Visual representation of LUBM query L2 with a large, complex BGP

```
    ?pub ub:publicationAuthor ?s1 .
    ?pub ub:publicationAuthor ?s2 .
  }
}
```

L3: An Associate Professor and a Full Professor who got their MS degrees from the same identified university, got their PhDs from two different identified universities and now are colleagues at the same university:

```
SELECT ?p1 ?uni ?n1 ?e1 ?ph1 ?res1 ?c ?pub1 ?pub2 ?p2 ?n2 ?e2 ?ph2 ?res2 WHERE {
  GRAPH ?g {
    ?p1 rdf:type ub:FullProfessor .
    ?p1 ub:undergraduateDegreeFrom <http://www.University584.edu> .
```

```
    ?p1 ub:mastersDegreeFrom <http://www.University584.edu> .
    ?p1 ub:doctoralDegreeFrom <http://www.University429.edu> .
    ?p1 ub:worksFor ?uni .
    ?p1 ub:name ?n1 .
    ?p1 ub:emailAddress ?e1 .
    ?p1 ub:telephone ?ph1 .
    ?p1 ub:researchInterest ?res1 .
    ?p1 ub:teacherOf ?c .
    ?p2 rdf:type ub:AssociateProfessor .
    ?p2 ub:undergraduateDegreeFrom <http://www.University584.edu> .
    ?p2 ub:mastersDegreeFrom <http://www.University584.edu> .
    ?p2 ub:doctoralDegreeFrom <http://www.University9999.edu> .
    ?p2 ub:worksFor ?uni .
    ?p2 ub:name ?n2 .
    ?p2 ub:emailAddress ?e2 .
    ?p2 ub:telephone ?ph2 .
    ?p2 ub:researchInterest ?res2 .
    ?p2 ub:teacherOf ?course2 .
    ?pub1 ub:publicationAuthor ?p1 .
    ?pub2 ub:publicationAuthor ?p2 .
  }
}
```

Small BGP queries L4-L12 are shown below.

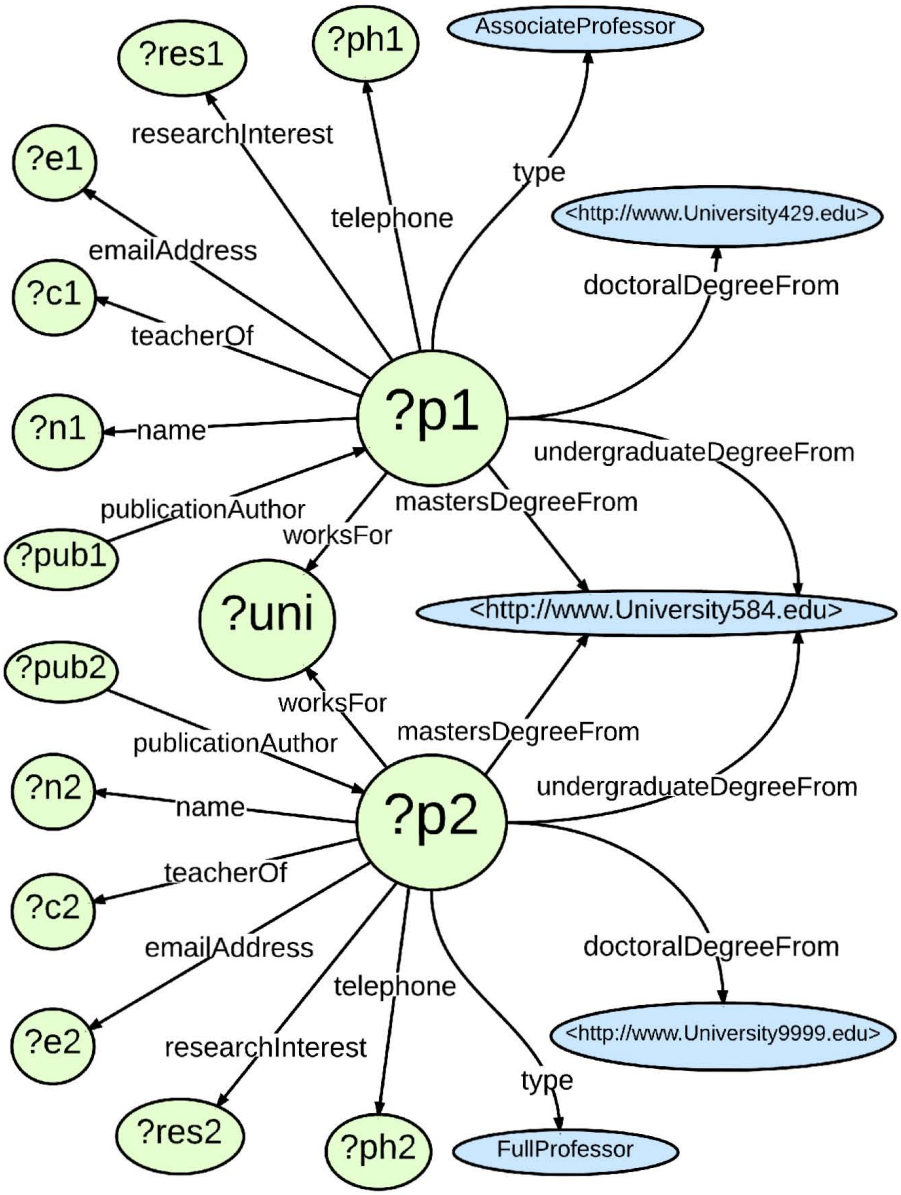


Figure 22. Visual representation of LUBM query L3 with a large, complex BGP

L4: All the graduate students, the undergraduate universities they attended and the graduate universities they currently attend along with the departments they are members of:

```
SELECT ?x ?y ?z WHERE {
  GRAPH ?g {
    ?z ub:subOrganizationOf ?y .
    ?y rdf:type ub:University .
    ?z rdf:type ub:Department .
    ?x ub:memberOf ?z .
    ?x rdf:type ub:GraduateStudent .
    ?x ub:undergraduateDegreeFrom ?y .
  }
}
```

L5: All the graduate students:

```
SELECT ?x WHERE {
  GRAPH ?g {
    ?x rdf:type ub:GraduateStudent .
  }
}
```

L6: Assistant Professors who are advisors to graduate students who are taking courses those professors are teaching:

```
SELECT ?x ?y ?z WHERE {
  GRAPH ?g {
    ?x rdf:type ub:GraduateStudent .
    ?y rdf:type ub:AssistantProfessor .
    ?z rdf:type ub:GraduateCourse .
    ?x ub:advisor ?y .
    ?y ub:teacherOf ?z .
    ?x ub:takesCourse ?z .
  }
}
```

L7: All the undergraduate students:

```
SELECT ?x WHERE {
  GRAPH ?g {
    ?x rdf:type ub:UndergraduateStudent .
  }
}
```

L8: The names of all the courses:

```
SELECT ?x WHERE {
  GRAPH ?g {
    ?x rdf:type ub:Course .
    ?x ub:name ?y .
```

```
  }
}
```

L9: All the undergraduate students who are taking courses being taught by their advisors who are Full Professors:

```
SELECT ?x ?y ?z WHERE {
  GRAPH ?g {
    ?y ub:teacherOf ?z .
    ?y rdf:type ub:FullProfessor .
    ?z rdf:type ub:Course .
    ?x ub:advisor ?y .
    ?x rdf:type ub:UndergraduateStudent .
    ?x ub:takesCourse ?z .
  }
}
```

L10: The emails of all the undergraduate students who are members of the department of an identified university:

```
SELECT ?x ?y ?z WHERE {
  GRAPH ?g {
    ?x rdf:type ub:UndergraduateStudent .
    ?y rdf:type ub:Department .
    ?x ub:memberOf ?y .
```

```
    ?y ub:subOrganizationOf <http://www.University0.edu> .
    ?x ub:emailAddress ?z .
  }
}
```

L11: All the Full Professors who work for the department of a specified university:

```
SELECT ?x ?y WHERE {
  GRAPH ?g {
    ?x rdf:type ub:FullProfessor .
    ?y rdf:type ub:Department .
    ?x ub:worksFor ?y .
    ?y ub:subOrganizationOf <http://www.University0.edu> .
  }
}
```

L12: All the undergraduate students who are members of the department of a university:

```
SELECT ?x ?y ?z WHERE {
  GRAPH ?g {
    ?x rdf:type ub:UndergraduateStudent .
    ?y rdf:type ub:University .
    ?z rdf:type ub:Department .
    ?x ub:memberOf ?z .
```

```
    ?z ub:subOrganizationOf ?y .
    ?x ub:undergraduateDegreeFrom ?y .
  }
}
```

A.2 BTC-2012 Queries

Large BGP queries B1 and B2 are shown below. Figures 23 and 24 shows the visual representation of each query's BGP. The common prefixes are listed below.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX geo: <http://aims.fao.org/aos/geopolitical.owl#>
PREFIX collect: <http://purl.org/collections/nl/am/>
PREFIX ore: <http://www.openarchives.org/ore/terms/>
PREFIX fbase: <http://rdf.freebase.com/ns/>
```

B1: Details about a product with specific characteristics (acquisition date and method, start and end production dates, etc.):

```
SELECT ?s1 ?o1 ?s2 WHERE {
  GRAPH ?g {
    ?s1 collect:acquisitionDate "1980-05-16" .
    ?s1 collect:acquisitionMethod collect:t-14382 .
    ?s1 collect:associationSubject ?o1 .
    ?s1 collect:contentMotifGeneral collect:t-8782 .
    ?s1 collect:creditLine collect:t-14773 .
    ?s1 collect:material collect:t-3249 .
    ?s1 collect:objectCategory collect:t-15606 .
    ?s1 collect:objectName collect:t-10444 .
    ?s1 collect:objectNumber "KA 17150" .
```

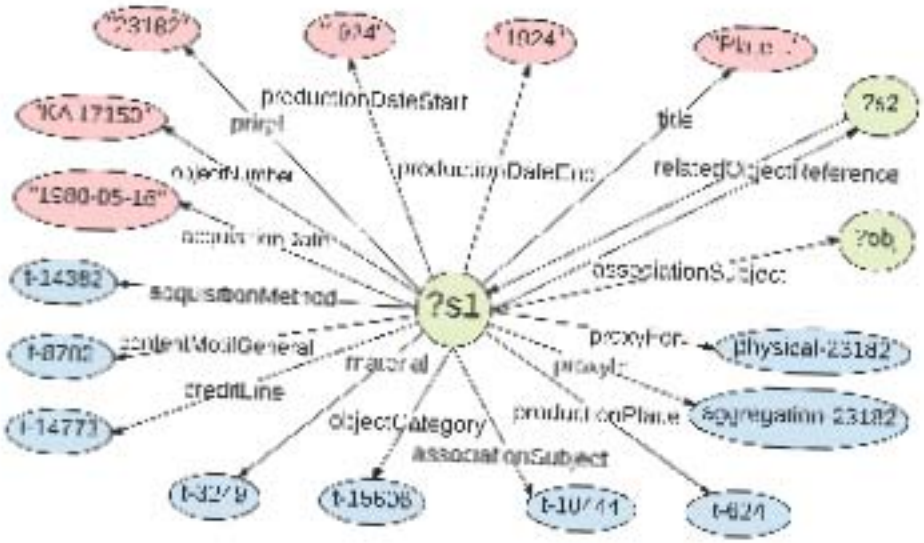


Figure 23. Visual representation of BTC query B1 with a large, complex BGP

```
    ?s1 collect:priref "23182" .
    ?s1 collect:productionDateEnd "1924" .
    ?s1 collect:productionDateStart "1924" .
    ?s1 collect:productionPlace collect:t-624 .
    ?s1 collect:title "Plate commemorating the first Amsterdam-Batavia flight"@en .
    ?s1 ore:proxyFor collect:physical-23182 .
    ?s1 ore:proxyIn collect:aggregation-23182 .
    ?s1 collect:relatedObjectReference ?s2 .
    ?s2 collect:relatedObjectReference ?s1 .
  }
}
```

B2: Two countries which are on two different specified continents, but share a border:

```
SELECT ?u ?un ?cnt1 ?ctry1 ?on1 ?cnt2 ?ctry2 ?on2 WHERE {
  GRAPH ?g {
    ?u geo:nameShortEN ?un .
    ?u geo:hasMember ?ctry1 .
    ?u rdf:type geo:economic_region .
    ?cnt1 geo:hasMember ?ctry1 .
    ?cnt1 rdf:type geo:geographical_region .
    ?cnt1 geo:nameShortEN "Africa"^^xsd:string .
    ?cnt2 geo:hasMember ?ctry2 .
    ?cnt2 rdf:type geo:geographical_region .
    ?cnt2 geo:nameShortEN "Asia"^^xsd:string .
    ?ctry1 geo:nameOfficialEN ?on1 .
    ?ctry1 geo:isInGroup ?u .
    ?ctry1 geo:isInGroup ?cnt1 .
    ?ctry1 geo:isInGroup geo:World .
    ?ctry1 rdf:type geo:self_governing .
    ?ctry1 geo:hasBorderWith ?ctry2 .
    ?ctry2 geo:nameOfficialEN ?on2 .
    ?ctry2 geo:isInGroup ?cnt2 .
    ?ctry2 geo:isInGroup geo:World .
    ?ctry2 rdf:type geo:self_governing .
    ?ctry2 geo:hasBorderWith ?ctry1 .
```

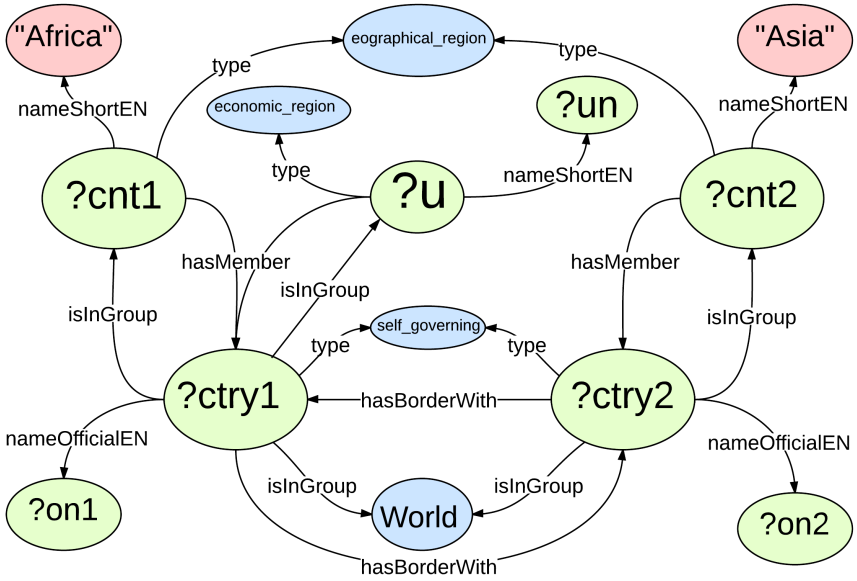


Figure 24. Visual representation of BTC query B2 with a large, complex BGP

```
  }
}
```

Small BGP queries B3-B7 are shown below.

B3: The names of films, actors in those films, and the release dates of the films:

```
SELECT ?fperf ?actor ?film ?name ?rel WHERE {
  GRAPH ?g {
    ?fperf fbase:film.performance.actor ?actor .
    ?fperf fbase:film.performance.film ?film .
    ?film fbase:type.object.name ?name .
```

```
    ?film fbase:film.film.initial_release_date ?rel .
  }
}
```

B4: More details about the films:

```
SELECT ?fperf ?actor ?film ?name ?rel ?lang ?gen WHERE {
  GRAPH ?g {
    ?fperf fbase:film.performance.actor ?actor .
    ?fperf fbase:film.performance.film ?film .
    ?film fbase:type.object.name ?name .
    ?film fbase:film.film.initial_release_date ?rel .
    ?film fbase:film.film.language ?lang .
    ?film fbase:film.film.genre ?gen .
  }
}
```

B5: More details about the films:

```
SELECT ?fperf ?actor ?film ?name ?rel ?lang ?gen ?star WHERE {
  GRAPH ?g {
    ?fperf fbase:film.performance.actor ?actor .
    ?fperf fbase:film.performance.film ?film .
    ?film fbase:type.object.name ?name .
    ?film fbase:film.film.initial_release_date ?rel .
```

```
    ?film fbase:film.film.language ?lang .
    ?film fbase:film.film.genre ?gen .
    ?film fbase:film.film.starring ?star .
  }
}
```

B6: Two people who live in the same place in Iraq:

```
SELECT ?p1 ?p2 ?p1n ?p2n ?loc WHERE {
  GRAPH ?g {
    ?p1 fbase:people.place_lived.person ?p1n .
    ?p1 fbase:people.place_lived.location ?loc .
    ?p2 fbase:people.place_lived.person ?p2n .
    ?p2 fbase:people.place_lived.location ?loc .
    ?loc fbase:location.location.containedby fbase:en.iraq .
  }
}
```

B7: Details about a location (people born there, events, country in which it is located):

```
SELECT ?s ?x ?y ?z ?w ?t WHERE {
  GRAPH ?g {
    ?s fbase:location.location.events ?x .
    ?s fbase:location.location.geolocation ?y .
```

```
    ?s fbase:location.location.people_born_here ?z .
    ?s fbase:location.location.people_born_here ?w .
    ?s fbase:location.location.containedby ?t .
  }
}
```

B8: Details about a city in the US (area code, time zone, postal code, population count, etc.):

```
PREFIX res: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>

SELECT ?city ?area ?code ?zone ?abstract ?postal ?water ?popu ?g
WHERE {
  GRAPH ?g {
    { ?city onto:areaLand ?area .
      ?city onto:areaCode ?code . }
    UNION
    { ?city onto:timeZone ?zone .
      ?city onto:abstract ?abstract . }
    ?city onto:country res:United_States .
    ?city onto:postalCode ?postal .
    OPTIONAL { ?city onto:areaWater ?water . }
    OPTIONAL { ?city onto:populationTotal ?popu . }
```

```
  }
}
```

B9: More details about the city:

```
PREFIX res: <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>

SELECT ?city ?area ?code ?zone ?abstract ?postal ?popu ?g
WHERE {
  GRAPH ?g {
    { ?city onto:areaLand ?area .
    ?city onto:areaCode ?code . }
    UNION
    { ?city onto:timeZone ?zone .
    ?city onto:abstract ?abstract . }
    ?city onto:country res:United_States .
    ?city onto:postalCode ?postal .
    OPTIONAL { ?city onto:populationTotal ?popu . }
  }
}
```

B10: Details about Brunei (population, homepage, latitude, longitude, etc.):

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

```
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT * WHERE {
  GRAPH ?g {
    ?var6 a <http://dbpedia.org/ontology/PopulatedPlace> .
    ?var6 <http://dbpedia.org/ontology/abstract> ?var1 .
    ?var6 rdfs:label ?var2 .
    ?var6 geo:lat ?var3 .
    ?var6 geo:long ?var4 .
    {
      ?var6 rdfs:label "Brunei"@en .
    } UNION {
      ?var5 <http://dbpedia.org/property/redirect> ?var6 .
      ?var5 rdfs:label "Brunei"@en .
    }
    OPTIONAL { ?var6 foaf:depiction ?var8 }
    OPTIONAL { ?var6 foaf:homepage ?var10 }
    OPTIONAL { ?var6 <http://dbpedia.org/ontology/populationTotal> ?var12 }
    OPTIONAL { ?var6 <http://dbpedia.org/ontology/thumbnail> ?var14 }
  }
}
```

B11: Thumbnails and homepages of all the people who have a Wikipedia article written about them:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX space: <http://purl.org/net/schemas/space/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpedia-prop: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT * WHERE {
  GRAPH ?g {
    ?var5 dbpedia-owl:thumbnail ?var4 .
    ?var5 rdf:type dbpedia-owl:Person .
    ?var5 rdfs:label ?var .
    ?var5 foaf:page ?var8 .
    OPTIONAL { ?var5 foaf:homepage ?var10 . }
  }
}
```

APPENDIX B

SPARQL GRAMMAR

```
Query => 'SELECT' Variables 'WHERE' '{' 'GRAPH' Variables
  '{' GroupGraphPattern '}' '}' ResultModifiers
GroupGraphPattern => BGP? ( GraphPatternNotTriples '.'? BGP? )*
GraphPatternNotTriples =>
  GroupOrUnionGraphPattern | OptionalGraphPattern | Filter
GroupOrUnionGraphPattern =>
  GroupGraphPattern ( 'UNION' GroupGraphPattern )*
OptionalGraphPattern => 'OPTIONAL' GroupGraphPattern
Filter => 'FILTER' Constraint
Constraint => Predicate | 'EXISTS' BGP | 'NOT EXISTS' BGP
```

REFERENCES

1. Data.gov: March, 2015. http://catalog.data.gov/dataset.
2. Data.gov.uk: March, 2015. http://data.gov.uk/data/search.
3. Google sparsehash: March, 2015. https://code.google.com/p/sparsehash/.
4. Have semantic technologies crossed the chasm yet?: March, 2015. https://semanticweb.com/have-semantic-technologies-crossed-the-chasm-yet_b16484.
5. Linked Data FAQ: March, 2015. http://linkeddata.org/faq.
6. Linking Open Gov. Data: March, 2015. http://logd.tw.rpi.edu/.
7. LUBM: February, 2015. http://swat.cse.lehigh.edu/projects/lubm/.
8. NYTimes.com Linked Open Data: February, 2015. http://data.nytimes.com.
9. Pfizer: December, 2014. https://semanticweb.com/tag/pfizer.
10. Resource Descrip. Framework: August, 2014. http://www.w3.org/RDF.
11. Seman. Web Challenge: August, 2015. http://challenge.semanticweb.org/.
12. Semantic Web Roadmap: September, 2014. http://www.w3.org/DesignIssues/Semantic.html.
13. SPARQL 1.1: November, 2014. http://www.w3.org/TR/sparql11-query/.
14. SPARQL Query Language for RDF: August, 2014. http://www.w3.org/TR/rdf-sparql-query/.
15. The Semantic Web Stack: February, 2015. http://en.wikipedia.org/wiki/Semantic_Web_Stack.

16. W3C Semantic Web Activity: January, 2015. http://www.w3.org/2001/sw/.
17. D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: A vertically partitioned DBMS for semantic web data management. *VLDB Journal*, 18(2):385–406, 2009.
18. Apache. Jena tdb. http://jena.apache.org/documentation/tdb/.
19. 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 2010 WWW Conference*, pages 41–50, 2010.
20. D. Beckett. Raptor. http://librdf.org/raptor/.
21. D. Beckett. Rasqal. http://librdf.org/rasqal/.
22. T. Berners-Lee. Linked Open Data. http://www.w3.org/2008/Talks/0617-lod-tbl/.
23. Bit.ly. Dablooms. https://github.com/bitly/dablooms.
24. C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The story so far. *Int. Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.
25. M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *Proc. of 2013 SIGMOD Conference*, pages 121–132, 2013.
26. M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. DOGMA: A disk-oriented graph matching algorithm for RDF databases. In *Proc. of ISWC '09*, pages 97–113, 2009.
27. A. Broder. On the resemblance and containment of documents. In *Proc. of the Compress. and Complex. of Sequences*, pages 21–29, 1997.

28. A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

29. 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*, pages 54–68, 2002.

30. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *Proc. of the 31st VLDB Conference*, pages 1216–1227, 2005.

31. O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. *CSSW*, pages 59–68, 2007.

32. O. Görlitz, M. Thimm, and S. Staab. SPLODGE: Systematic Generation of SPARQL Benchmark Queries for Linked Open Data. *International Semantic Web Conference*, 7649(Chapter 8):116–132, 2012.

33. S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. In *Practical and Scalable Semantic Systems*, 2003.

34. T. H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *WebDB (Informal Proceedings)*, pages 129–134, 2000.

35. T. H. Haveliwala, A. Gionis, D. Klein, and P. Indyk. Evaluating strategies for similarity search on the Web. In *Proc. of the 11th WWW Conference*, pages 432–442, 2002.

36. K. Hose, R. Schenkel, M. Theobald, and G. Weikum. Database Foundations for Scalable RDF Processing. *Reasoning Web*, 6848(Chapter 4):202–249, 2011.

37. J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *Proc. of VLDB Endow.*, 4(11):1123–1134, 2011.

38. P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. of the 13th ACM STOC*, pages 604–613, 1998.

39. P. Jaccard. *Distribution de la Flore Alpine: dans le Bassin des dranses et dans quelques régions voisines*. Rouge, 1901.

40. Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *The VLDB Journal*, pages 1–25, 2014.

41. A. Kiryakov, B. Bishoa, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. The features of bigowlim that enabled the bbcs world cup website. In *Workshop on Semantic Data Management*, 2010.

42. A. J. Max Schmachtenberg, Christian Bizer and R. Cyganiak. Linking Open Data cloud diagram, 2014. http://lod-cloud.net.

43. E. F. Moore. *The shortest path through a maze*. Bell Telephone System., 1959.

44. M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. Dbpedia sparql benchmark–performance assessment with real queries on real data. In *The Semantic Web–ISWC 2011*, pages 454–469. Springer, 2011.

45. M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. Usage-Centric Benchmarking of RDF Triple Stores. *AAAI 2012*, 2012.

46. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

47. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.

48. T. Neumann and G. Weikum. xRDF3X: Fast Querying, High Update Rates, and

Consistency for RDF Databases. *Proc. VLDB Endow.*, 2010.

49. A. Owens, A. Seaborne, and N. Gibbins. Clustered TDB: A Clustered Triple Store for Jena - ePrints Soton. 2008.
50. M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR 15-81, Harvard University, 1981.
51. S. Sakr and G. Al-Naymat. Relational processing of RDF queries: a survey. *ACM SIGMOD Record*, 38(4):23–28, June 2010.
52. V. Slavov, A. Katib, and P. Rao. A tool for Internet-scale cardinality estimation of XPath queries over distributed semistructured data. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 1270–1273, 2014.
53. V. Slavov, A. Katib, P. Rao, S. Paturi, and D. Barenkala. Fast Processing of SPARQL Queries on RDF Quadruples. In *Proc. of WebDB '14*, pages 1–6, 2014.
54. V. Slavov and P. Rao. Towards internet-scale cardinality estimation of xpath queries over distributed xml data. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB). Athens, Greece*, 2011.
55. V. Slavov and P. Rao. A gossip-based approach for Internet-scale cardinality estimation of XPath queries over distributed semistructured data. *The VLDB Journal — The International Journal on Very Large Data Bases*, 23(1), Feb. 2014.
56. O. Software. Virtuoso. http://virtuoso.openlinksw.com.
57. J. H. Tim Berners-Lee and O. Lassila. The Semantic Web. http://www.scientificamerican.com/article/the-semantic-web/.
58. O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: a graph based RDF

index. In *Proc. of the 22nd National Conf. on Artificial Intelligence*, pages 1465–1470, 2007.

59. C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for Semantic Web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, 2008.
60. Wikipedia. LSH Applications. http://en.wikipedia.org/wiki/Locality-sensitive_hashing#Applications.
61. K. Wilkinson. Jena property table implementation. In *SSWS 2006*, pages 35–46, Athens, GA, 2006.
62. K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proc. of SWDB'03*, pages 131–150, 2003.
63. P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: A fast and compact system for large scale RDF data. *Proc. VLDB Endow.*, 6(7):517–528, 2013.
64. K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for Web Scale RDF data. *Proc. VLDB Endow.*, 6(4):265–276, Feb. 2013.
65. L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL queries via subgraph matching. *Proc. VLDB Endow.*, 4:482–493, May 2011.

VITA

Vasil Slavov received a M.S. in Computer Science from the University of Missouri-Kansas City in 2012. He received a B.A. in Computer Science and Mathematics from William Jewell College in 2005 and worked as a Network Administrator at the Kansas City Art Institute until 2011. He joined the Ph.D. program at UMKC full-time in 2011. His research interests are in the area of scalable RDF query processing, RDF quadruples, federated SPARQL query processing, and cardinality estimation of queries over semi-structured data using gossip-based algorithms. He joined Bloomberg L.P. in New York, NY in 2015 as a Software Engineer.