ARTICLE IN PRESS

Web Semantics: Science, Services and Agents on the World Wide Web ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Web Semantics: Science, Services and Agents on the World Wide Web

journal homepage: www.elsevier.com/locate/websem

RIQ: Fast processing of SPARQL queries on RDF quadruples

Anas Katib, Vasil Slavov, Praveen Rao *

Department of Computer Science and Electrical Engineering, University of Missouri-Kansas City, Kansas City, MO, USA

ARTICLE INFO

Article history:
Received 9 April 2015
Received in revised form
31 January 2016
Accepted 16 March 2016
Available online xxxx

Keywords:
RDF
Quadruples
SPARQL
Query processing
Knowledge graphs

ABSTRACT

In this paper, we address the problem of fast processing of SPARQL queries on a large RDF dataset, where the RDF statements are quadruples (or quads). Quads can capture provenance or other relevant information about facts. This is especially powerful in modeling knowledge graphs, which are becoming increasingly important on the Web to provide high quality search results to users. We propose a new approach called RIQ that employs a *decrease-and-conquer* strategy for fast SPARQL query processing. Rather than indexing the entire RDF dataset, RIQ identifies groups of similar RDF graphs and creates indexes on each group separately. It employs a new vector representation for RDF graphs and locality sensitive hashing to construct the groups efficiently. It constructs a novel filtering index on the groups and compactly represents the index as a combination of Bloom and Counting Bloom Filters. During query processing, RIQ employs a streamlined approach. It constructs a query plan for a SPARQL query (containing one or more graph patterns), searches the filtering index to quickly identify candidate groups that may contain matches for the query, and rewrites the original query to produce an optimized query for each candidate. The optimized queries are then executed using an existing SPARQL processor that supports quads to produce the final results. We conducted a comprehensive evaluation of RIQ using a real and synthetic dataset, each containing about 1.4 billion quads. Our results show that RIQ can outperform its competitors designed to support named graph queries on RDF quads (e.g., Jena TDB and Virtuoso) for a variety of queries.

1. Introduction

The Resource Description Framework (RDF) is a standard model for data representation and interchange on the Web [1]. Today, RDF uses IRIs to name entities and their relationships. It enables easy merging of different data sources. While RDF was introduced in the late 90s as the data model for the Semantic Web, only in recent years, it has gained popularity on the Web. For example, Linked Data [2] exemplifies the use of RDF on the Web to represent different knowledge bases (*e.g.*, DBpedia [3]). Another example is Wikidata [4], a sister project of Wikipedia, which publishes facts in RDF. Advanced RDF technologies provide the ability to conduct semantic reasoning in domains such as biopharmaceuticals, defense and intelligence, and healthcare. Several companies have adopted Semantic Web technologies for different use cases such as data aggregation (*e.g.*, Pfizer [5]), publishing datasets on the Web and providing better quality search results (*e.g.*, Newsweek, BBC, The New York Times, Best Buy) [6].

Another important use case of RDF is in the representation of knowledge graphs, which are emerging as a key resource for companies like Google [7], Facebook [8], and Microsoft [9] to provide higher quality search results and recommendations to users. Essentially, a knowledge graph is a collection of entities, their properties, and relationships among entities. Using SPARQL [10], queries can be posed on these knowledge graphs.

In RDF, a fact or assertion is represented as a (subject, predicate, object) triple. A set of RDF triples can be modeled as a directed, labeled graph. A triple's subject and object denote the source and sink vertices, respectively, and the predicate is the label of the edge from the source to the sink. An RDF quad is denoted by a (subject, predicate, object, context). The context (a.k.a. graph name) is used to capture the provenance or other relevant information of a triple. This is especially powerful in modeling the facts in a knowledge graph. Moreover, there are datasets and knowledge bases on the Web such as Billion Triples Challenges [11], Linking Open Government Data (LOGD) [12], and Yago [13] which contain over a billion quads. One can view these datasets as a collection of RDF named graphs. Using SPARQL's GRAPH keyword [10], a query can be posed on RDF named graphs to match a specific graph pattern within any single RDF graph.

* Corresponding author.
E-mail addresses: anaskatib@mail.umkc.edu (A. Katib), vgslavov@mail.umkc.edu (V. Slavov), raopr@umkc.edu (P. Rao).

http://dx.doi.org/10.1016/j.websem.2016.03.005
1570-8268/

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

res:Oswego onto:areaLand "5438975.0317056"^^<http://www.w3.org/2001/XMLSchema#double> <http://dbpedia.org/data/Oswego.xml> .
res:Oswego onto:areaCode "620"@en <http://dbpedia.org/data/Oswego.xml> .
res:Oswego onto:isPartOf res:Labette_County,_Kansas <http://dbpedia.org/data/Oswego.xml> .
res:Oswego onto:country res:United_States <http://dbpedia.org/data/Oswego.xml> .
res:Oswego onto:postalCode "67356"@en <http://dbpedia.org/data/Oswego.xml> .
res:Oswego onto:utcOffset "-6"@en <http://dbpedia.org/data/Oswego.xml> .
res:Oswego onto:utcOffset "-5"@en <http://dbpedia.org/data/Oswego.xml> .
res:Oswego onto:areaWater "0"^^<http://www.w3.org/2001/XMLSchema#double> <http://dbpedia.org/data/Oswego.xml> .                    G1

res:Salamiou onto:abstract "Salamiu - wie\u015B na Cyprze, w dystrykcie Pafos."@pl <http://dbpedia.org/data/Salamiou.xml> .
res:Salamiou onto:timeZone res:Eastern_European_Summer_Time <http://dbpedia.org/data/Salamiou.xml> .
res:Salamiou onto:isPartOf res:Paphos_District <http://dbpedia.org/data/Salamiou.xml> .
res:Salamiou onto:country res:Cyprus <http://dbpedia.org/data/Salamiou.xml> .
res:Salamiou onto:postalCode "6211"@en <http://dbpedia.org/data/Salamiou.xml> .
res:Salamiou onto:utcOffset "+3"@en <http://dbpedia.org/data/Salamiou.xml> .
res:Salamiou onto:utcOffset "+2"@en <http://dbpedia.org/data/Salamiou.xml> .
res:Salamiou onto:populationTotal "255"^^<http://www.w3.org/2001/XMLSchema#integer> <http://dbpedia.org/data/Salamiou.xml> .   G2
```

Fig. 1. Dataset D containing RDF quads.

The popularity of the RDF data model coupled with the availability of very large RDF datasets continues to pose interesting technical challenges for storing, indexing, and query processing of RDF data. In this paper, we address the problem of fast processing of SPARQL queries on RDF quads. In recent years, there has been a flurry of interest within the database community to develop scalable techniques for indexing and query processing of large RDF datasets. Several techniques have been proposed for RDF datasets containing triples [14–21], where each triple consists of a subject, predicate, and an object. One may wonder if we can simply ignore the context in a quad and use any of the previous approaches for processing a query with the `GRAPH` keyword. Unfortunately, this may produce incorrect results, because subpatterns of a graph pattern may match RDF terms in different graphs. Furthermore, none of these approaches has investigated how large, complex graph patterns (*e.g.*, containing undirected cycles) in SPARQL queries can be processed efficiently. Evidently, RDF-3X [16], a popular scalable approach for a local environment, yields poor performance when SPARQL queries containing large, complex graph patterns are processed over large RDF datasets [22]. This is because of the large number of join operations that must be performed to process a query. We argue that, on RDF datasets containing billions of quads, any approach that first finds matches for subpatterns in a large graph pattern and then employs join operations to merge partial matches will face a similar limitation.

Motivated by the above reasons, we developed a new tool called RIQ (**R**DF **I**ndexing on **Q**uads) for fast processing of SPARQL queries on RDF quads. The salient features of RIQ are summarized below:

- RIQ adopts a new vector representation for RDF graphs and graph patterns in SPARQL queries. This representation captures the properties of the triples in an RDF graph and triple patterns in a query. It facilitates grouping similar RDF graphs using locality sensitive hashing [23] and building a novel filtering index for efficient query processing. RIQ uses a combination of Bloom Filters and Counting Bloom Filters to compactly store the filtering index. In addition to the filtering index, each group of similar RDF graphs is indexed separately rather than constructing a single index on the entire collection of RDF graphs.
- RIQ employs a streamlined approach to efficiently process a SPARQL query via the *decrease-and-conquer* strategy. Using the filtering index, RIQ quickly identifies candidate groups of RDF graphs that may contain a match for the query. It methodically rewrites the original query and executes optimized queries on the candidates using a conventional SPARQL processor that supports quads (*e.g.*, Jena TDB [24]).
- RIQ achieved high performance on a real-world and synthetic dataset, each containing about 1.4 billion quads, on a variety of SPARQL queries. It yielded superior performance for high selectivity queries that matched a small fraction of the named graphs in a dataset, when I/O was the dominating factor.

A preliminary version of this work appeared in the 17th International Workshop on the Web and Databases (WebDB) 2014 [22].

The rest of the paper is organized as follows. Section 2 provides the background on RDF and SPARQL. Section 3 describes the related work and the motivation of our work. Section 4 describes the novel design of RIQ including the new vector representation of RDF graphs and graph patterns, filtering index construction, and the query processing approach. Section 5 presents the performance evaluation results and comparison of RIQ with its competitors. Finally, we provide our concluding remarks in Section 6.

2. Background and preliminaries

In this section, we provide a brief background on RDF and SPARQL. After that, we describe popular techniques based on hashing that underpin the design of RIQ.

2.1. RDF and SPARQL

The RDF data model provides a simple way to represent any assertion as a (subject, predicate, object) triple. A collection of triples can be modeled as a directed, labeled graph. A triple can be extended with a graph name (or context) to form a quad. Quads with the same context belong to the same RDF graph.

Using SPARQL, one can express complex graph pattern queries on RDF graphs. A triple pattern contains variables (prefixed by `?`) and constants. A Basic Graph Pattern (BGP) in a query combines a set of triple patterns. During query processing, the variables in a BGP are bound to RDF terms in the data, *i.e.*, the nodes in the same RDF graph, via subgraph matching [10]. Common variables within a BGP or across BGPs denote a join operation on the variable bindings of triple patterns. `UNION` combines bindings of multiple graph patterns; `OPTIONAL` allows certain patterns to have empty bindings; `FILTER EXISTS/NOT EXISTS` tests for existence/non-existence of certain graph patterns. The variable `?g` will be bound to the contexts of those RDF graphs that contain a match for the entire set of graph patterns and predicates, if any, inside the `GRAPH` block.

Example 1. Consider the dataset D shown in Fig. 1, which contains two RDF graphs G_1 and G_2. Consider a query Q shown in Fig. 2. It has five BGPs. Consider the pattern BGP_1 in Q. The bindings for the variable `?city` in the triple pattern `?city onto:areaLand ?area` are joined with those for `?city` in `?city onto:areaCode ?code`. If Q is executed on D, `?g` will be bound only to the context of G_1, *i.e.*, `<http://dbpedia.org/data/Oswego.xml>`. Note that BGP_3 does not have a match in G_2 as the only country mentioned in G_2 is Cyprus.

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

```
SELECT * WHERE {
  GRAPH ?g {
    { ?city onto:areaLand ?area .                                  ------► BGP1
      ?city onto:areaCode ?code . }
    UNION
    { ?city onto:timeZone ?zone .                                  ------► BGP2
      ?city onto:abstract ?abstract . }
    ?city onto:country res:United_States .                         ------► BGP3
    ?city onto:postalCode ?postal .
    FILTER EXISTS { ?city onto:utcOffset ?offset . }               ------► BGP4
    OPTIONAL { ?city onto:populationTotal ?popu . }                ------► BGP5
  }}
```

Fig. 2. Query Q.

2.2. Popular techniques based on hashing

2.2.1. Rabin's fingerprinting

Michael Rabin (1981) proposed a fingerprinting technique to efficiently generate short hash codes (of configurable length) for arbitrary length bit strings [25]. If the hash codes of two bit strings are different, then the bit strings are definitely different. This technique is popularly called Rabin's fingerprinting and provides an efficient method for string matching.

Suppose a data item to be hashed is represented using its bit string representation. This corresponds to a polynomial in Galois Field 2 with coefficients 0 or 1. Let us denote this polynomial by p. We define $h(\cdot) = p \bmod r$, where r is an irreducible polynomial in Galois Field 2 picked at random. This hash function $h(\cdot)$ has low degree of collision. Suppose we select an irreducible polynomial of degree $d-1$. We can generate d-bit hash values using $h(\cdot)$. Given two data items x and y, s.t. $x \neq y$, $P(h(x) = h(y)) \leq \frac{max(|x|,|y|)}{2^{d-1}}$, where $|\cdot|$ denotes the number of bits in a data item [26]. As an example, suppose we choose r to be an irreducible polynomial of degree 31 and generate 32-bit hash values. Suppose each data item requires at most 2^{20} bits. Then the probability of collision is less than 2^{-11}, which is very low in practice.

2.2.2. Locality sensitive hashing

Locality sensitive hashing (LSH) was introduced by Indyk and Motwani [23] for approximate nearest neighbor search on high-dimensional data. Intuitively, in LSH, data items are hashed using several hash functions such that each hash function is more likely to produce collisions on data items that are similar to each other than on those that are dissimilar. A benefit of LSH is that it provides a fast probabilistic method of grouping similar data items in a dataset without computing the exact similarity between every pair of data items in it. Over the years, LSH has been employed in many domains, including indexing high dimensional data and similarity searching [27,28], similarity searching over web data [29] and in P2P networks [29,30], ranges queries in P2P networks [31], cardinality estimation over distributed XML documents [32], and so forth.

Consider two sets S_1 and S_2 whose similarity based on the Jaccard index is given by $p = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$. By using LSH (defined on sets) [29], we can estimate if these two sets are similar (or dissimilar) given a similarity cutoff without actually computing p. As a result, we can quickly construct groups of similar sets in a large database without computing the Jaccard index of every pair of sets.

Let us denote the LSH operation on a set S by $\mathsf{LSH}_{k,l,m}(S)$, where k, l, and m are configurable parameters. $\mathsf{LSH}_{k,l,m}(S)$ will produce k hash values, each in the range $[0, m-1]$. We compute $\mathsf{LSH}_{k,l,m}(S)$ as follows [29]: Consider a linear hash function of the form $(ax + b) \bmod u$, where x is the item to be hashed, u is a prime, and a and b are integers such that $0 < a < u$ and $0 \leq b < u$. We pick $k \times l$ such random linear hash functions by choosing a and b at random. Let us denote each hash function by $\hbar_{ij}$. Compute $g_{ij}(S) = \min\{\hbar_{ij}(x)\}$ over all items in the set. Create k groups of l values using the output of all the $g_{ij}(\cdot)$ functions, where $1 \leq i \leq k$ and $1 \leq j \leq l$. For each group of l values, concatenate these values and hash them to the range $[0, m-1]$ (*e.g.*, using Rabin's fingerprinting). As there are k groups, $\mathsf{LSH}_{k,l,m}(S)$ produces k hash values for S.

It has been shown that for two sets S_1 and S_2 with Jaccard index p, $\Pr[g_{ij}(S_1) = g_{ij}(S_2)] = p$. Also, the probability that $\mathsf{LSH}_{k,l,m}(S_1)$ and $\mathsf{LSH}_{k,l,m}(S_2)$ have at least one hash value identical (or collision) is $1-(1-p^l)^k$. (The above properties also hold true for multisets.)

The value of m can be chosen so that the hash values are represented using 32-bit integers. The values of k and l are typically chosen based on the desired similarity cutoff between sets and the computational cost. If we fix $k = 100$ and $l = 10$, the cutoff is close to 0.4 using a total of 1000 linear hash functions. That is, two sets with similarity less than 0.4 have almost zero probability of producing any collision. If $l = 20$, the cutoff increases to 0.6 using a total of 2000 linear hash functions.

2.2.3. Bloom and counting bloom filters

A Bloom filter (BF) is a randomized data structure to compactly represent a set of items in order to support membership queries. A BF maintains an array of n bits, which are initialized to 0, and uses m independent hash functions with range $[0, n-1]$. To insert a data item into this BF, we apply the m hash functions to get the bit positions in the array. These bits are set to 1. To test if an item exists in the filter, we apply the same m hash functions and test if the corresponding bit positions are set to 1. As a result, false positives may occur. If at least one of those bit positions is 0, then the item is definitely not present in the filter. A Counting Bloom filter (CBF) maintains t-bit counters instead of single bits and can be used to compactly represent multisets. It has been shown that 4-bit counters should be sufficient in most applications based on how likely is a counter to overflow [33]. Based on how many items need to be inserted into a filter, a BF or CBF can be configured to achieve a particular false positive rate [33].

3. Related work and motivation

3.1. RDF query processing

Today, there are a number of open-source and commercial tools for storing and querying RDF graphs (*e.g.*, Jena [34,35], Sesame [36], Virtuoso [37], Garlik 4store [38], AllegroGraph [39], Mulgara [40], YARS2 [41], Kowari [42], 3Store [43], Bigdata(R) [44], Oracle 11g RDF [45,46], Neo4j RDF [47]). These tools either store and process RDF in main-memory, use an RDBMS, or a native RDF database. The popular approach has been to use relational database systems for storing, indexing, and querying RDF [34,35,43,36,46,48,49]. Some have attempted a graph based approach of storing and querying RDF [50,51]; a few have taken a path-based approach by storing subgraphs in relational tables [52,53]. But these graph and path-based techniques were evaluated on small RDF datasets. A few of the proposed techniques are main memory based RDF stores [54,55].

Unfortunately, the cost of self-joins on a single (triples) table became a serious bottleneck. Later, Abadi et al. proposed the idea of vertically partitioning the property tables [56] and used a column-oriented DBMS to achieve an order of magnitude performance improvement over previous techniques [14]. Recently, Neumann et al. developed RDF-3X [16] that builds exhaustive indexes on the six permutations of (s, p, o) triples. RDF-3X significantly outperformed the vertical partitioning approach. It uses a new join ordering method based on selectivity estimates and builds compressed indexes. Weiss et al. [15] developed Hexastore that also builds exhaustive indexes. However, Hexastore suffers from large index sizes due to lack of compression. Atre et al. [17] developed BitMat to overcome the overhead of large intermediate join results for queries containing low selectivity triple patterns. BitMat performs in-memory processing of compressed bit matrices

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

during query processing. Leeka et al. [57] proposed RQ-RDF-3X by adding quad indexes to RDF-3X to handle queries on quads and reification statements. As stated by the authors, their implementation did not support named graph queries.

More recently, Bornea et al. [19] developed DB2RDF by using an RDBMS to store and query RDF data. By storing the predicate–object pairs of each subject in the same row of the relational table, they reduced the number of joins required for star-shaped BGPs. DB2RDF maintains only subject and object indexes and employs a novel SPARQL-to-SQL translation technique for generating optimized queries. Yuan et al. [20] developed TripleBit, which uses a compact storage scheme for RDF data by representing triples via a Triple Matrix. For each predicate, TripleBit maintains SO and OS ordered buckets. Using a collection of indexes and optimal join ordering, it reduces the size of the intermediate results during query processing.

A few approaches exploit the graph properties of RDF data for indexing and query processing [58–62]. These techniques, however, have been tested only on small RDF datasets containing less than 50 million triples.

Recently, a few distributed and parallel SPARQL query processing approaches were proposed for datasets containing RDF triples [18,21,63–65]. Huang et al. [18] proposed a parallel SPARQL query processing approach by partitioning graphs on vertices and placing triples on different machines. Using n-hop replication of triples in partitions, they avoid communication between partitions during query processing. Later, Trinity.RDF was developed [21], where RDF graphs are stored natively using Trinity, a distributed in-memory key–value store. Using graph exploration and novel optimization techniques, the size of intermediate results is reduced leading to faster query execution. Recently, H2RDF+ [63] was proposed and builds eight indexes using HBase. It uses Hadoop to perform sort-merge joins during query processing. TriAD [64] is another approach and uses asynchronous inter-node communication for scalable SPARQL query processing. It outperforms distributed RDF query engines that rely on Hadoop to perform joins during query processing. DREAM [65] proposes the Quadrant-IV paradigm and partitions queries instead of data and selects different number of machines to execute different SPARQL queries based on their complexity. It employs a graph-based query planner and a cost model to outperform its competitors.

Note that RIQ is a centralized approach for efficient query processing on RDF datasets containing over a billion quads.

3.2. Pattern matching techniques on graphs

A few research attempts have been made for pattern matching in directed graphs. Chen et al. studied pattern matching on DAGs [66] by adapting an XML pattern matching algorithm called TwigStack [67]. Cheng et al. proposed a technique for directed graphs called R-join to find all occurrences of a graph pattern in a large data graph [68]. Zou et al. [69] developed a method for finding graph pattern matches by considering weight constraints on the query edges.

Much research has been done on developing efficient methods for processing exact subgraph matching queries on undirected graphs. These methods built a filtering index to quickly identify candidates followed by the verification phase (*e.g.*, using Ullmann's algorithm [70]) to discard false matches. They differ in how they select certain features of the input graph(s) for building the filtering index. For example, GraphGrep [71] built an index by selecting paths up to a certain length. On the other hand, gIndex [72] and FG-index [73] used discriminative frequent substructures (*i.e.*, subgraphs) in the data as the indexing feature. TreePi [74] and Tree + Δ [75] used frequent trees in graphs as the indexing feature to reduce the index size and construction cost. Later, QuickSI [76] was proposed to reduce the cost of the verification phase and leveraged a feature-based index (*i.e.*, using trees) to speed up the filtering phase.

Some of the techniques avoided frequent pattern mining and developed transformations on the input graphs for building a filtering index. For example, C-tree [77] constructed a hierarchical index by computing graph closures. GDIndex [78] proposed the use of graph decomposition to index graphs. GCoding [79] mapped the structure information of graphs into a numerical space using vertex signatures and graph codes, which are then used for indexing. GiS [80,81] mapped graphs to signatures using line graphs. These signatures were then organized by a hierarchical index for fast filtering.

While the above techniques are designed for graph pattern matching, they do not support SPARQL queries. On the other hand, RIQ aims to speed up the processing of SPARQL queries using a novel filtering index, which organizes summaries of RDF graphs designed specifically for BGP matching.

3.3. Motivation

The motivation for our work stems from three key observations: First, knowledge graphs are becoming a powerful resource for users of the World Wide Web. RDF quads can aptly model the facts in a knowledge graph, and SPARQL can be used to pose rich queries on RDF quads. Second, the approaches discussed in Section 3.1 were designed to process RDF datasets containing triples. Simply ignoring the context in an RDF quad and using an existing approach designed for triples may produce incorrect results due to bindings for a BGP from different graphs. For example, consider the two quads: `<a> <b> <c> <g1>. <a> <b> <e> <g2>.` If we use a technique designed to process triples for the query `SELECT ?x WHERE { GRAPH ?g { ?x <b> <c>. ?x <b> <e>. } }` on these quads. Then, `?x` will be bound to `<a>` as triples '`<a> <b> <c>`' and '`<a> <b> <e>`' will be treated as part of the same graph. In reality, the correct evaluation of this query should produce no results. Third, most of the queries tested by these approaches contain BGPs with a modest number of triples patterns (at most 8). None of them have investigated how to efficiently process SPARQL queries with large and complex BGPs (*e.g.*, containing undirected cycles[1]). A few examples are shown in Appendix B.

4. The design of RIQ

In this section, we present the novel design of RIQ (RDF Indexing on Quadruples).

4.1. Key components of RIQ

Fig. 4 shows the architecture of RIQ. RIQ is designed to deal with SPARQL `SELECT` queries on named graphs that conform to a subset of the grammar of the SPARQL query language [10]. These queries can contain constructs like `UNION`, `OPTIONAL`, and `FILTER`. Since the grammar of the SPARQL language is intricate, we employ a simpler grammar based on the seminal work of Pérez et al. [82]. The grammar of SPARQL queries is shown in Fig. 3. Let I, L, and V denote the set of all possible IRIs, literals, and variables, respectively. The terminal symbol `TriplePattern` $\in (I \cup V) \times (I \cup V) \times (I \cup L \cup V)$. We use parenthesis in the grammar to explicitly specify the precedence and associativity of the operators as suggested by Pérez et al. [82]. According to the

[1] Here is an example: {?a <p> ?b. ?b <q> ?c. ?a <r> ?c.}.

```
Query => 'SELECT' Variables 'WHERE' '{' 'GRAPH' Variables '{' Pattern '}' '}'
Pattern => ( Pattern '.' Pattern ) | ( Pattern 'UNION' Pattern ) | ( Pattern 'OPTIONAL' Pattern ) |
      ( Pattern 'FILTER' Constraint ) | TriplesBlock
Constraint => '(' Expression ')' | 'EXISTS' Pattern | 'NOT EXISTS' Pattern
TriplesBlock => TriplePattern ( '.' TriplePattern )*
```

Fig. 3. Grammar of SPARQL queries.

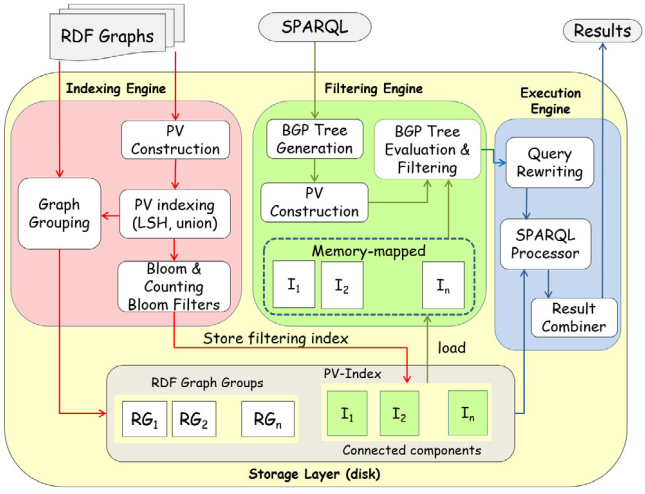


Fig. 4. Overview of RIQ.

SPARQL specification [10], `FROM NAMED` may be used in a SPARQL query to specify the dataset that should be used for matching on named graphs. Because our goal is to execute a SPARQL `SELECT` query on a dataset with millions of named graphs, we ignore this specification as it would be impossible to specify all the graphs in the query. Hence, our grammar does not contain `FROM NAMED`.

The key components of RIQ are the Indexing Engine, the Filtering Engine, and the Execution Engine. The Indexing Engine transforms an RDF graph into its vector representation, constructs a filtering index based on the vector representation by creating groups of similar RDF graphs, and builds a separate index on each group. The Filtering Engine generates a query plan for a SPARQL query, constructs the vector representation of each BGP in the query, and identifies, using the filtering index, candidate groups that may contain a match for the query. The Execution Engine rewrites the query methodically to generate an optimized query for each candidate group. It executes the optimized queries using an existing SPARQL processor that supports quads to produce the final output.

4.2. Indexing RDF data

We introduce a new vector representation for RDF graphs and BGPs, which will allows us to capture the properties of the triples and triple patterns in them. This vector representation plays a key role in the construction of an effective filtering index, where similar RDF graphs will be grouped together.

4.2.1. Essential transformations

To begin with, we define two transformations: one for a triple in an RDF graph and the other for a triple pattern in a BGP. Let $\mathbb{P} = \{SPO, SP?, S?O, ?PO, S??, ?P?, ??O\}$ be a set of canonical patterns. Let R denote the set of all possible RDF triples. We denote the transformation on a triple $(s, p, o) \in R$ by $f_D : \mathbb{P} \times R \rightarrow O_D$, where the range O_D is show in Table 1 for each canonical pattern. Note that O_D resembles triple patterns (variable names excluded) that can appear in a BGP.

Next, we denote a transformation $f_Q : T \rightarrow \mathbb{P} \times O_Q$, where T denotes the set of triple patterns that can appear in a query.

Table 1
Transformations in RIQ.

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

The range $\mathbb{P} \times O_Q$ is shown in Table 1 and identifies the canonical pattern for a given triple pattern. Although the triple pattern 's p o' has no variables, it is still a valid triple pattern in a BGP.[2]

The transformations f_D and f_Q allow us to map a triple in the data and a triple pattern in a query to a common plane of reference. This will enable us to quickly test *if a triple pattern in a BGP has a match in the data*.

4.2.2. Pattern vectors

Given an RDF graph with context c, we map it into a vector representation called a Pattern Vector (PV) and denote it by $\overline{V_c}$. Essentially, $\overline{V_c} = (V_{c,SPO}, V_{c,SP?}, V_{c,S?O}, V_{c,?PO}, V_{c,S??}, V_{c,?P?}, V_{c,??O})$, where each $V_{c,r}$ denotes the vector constructed for $r \in \mathbb{P}$. We assume a hash function $\mathbb{H} : B \rightarrow \mathbb{Z}^*$, where B denotes a bit string and the range is the set of non-negative integers. Now, we construct $\overline{V_c}$ as follows: Initially, each $V_{c,r}$ is empty. Given a quad (s, p, o, c) in the graph, for each $r \in \mathbb{P}$, we compute $\mathbb{H}(f_D(r, (s, p, o)))$ and insert it into $V_{c,r}$. We perform this computation on every quad in the graph to generate $\overline{V_c}$. Each $V_{c,r}$ is finally sorted, which will speed up the construction of the filtering index. Algorithm 1 shows the steps involved. $\overline{V_c}$ requires space linear in the number of quads in the graph. Note that each $V_{c,r}$ is a (dynamic) array and not a hash table.

Algorithm 1 PV construction for an RDF graph

Input: An RDF graph G with context c
Output: PV $\overline{V_c}$

```
1: for each (s, p, o, c) ∈ G do
2:     for each r ∈ ℙ do
3:         insert ℍ(f_D(r, (s, p, o))) into V_{c,r}
4: for each r ∈ ℙ do
5:     sort V_{c,r}
6: return V̄_c
```

Our hash function $\mathbb{H}$ is based on Rabin's fingerprinting technique, which was presented in Section 2.2.1. While IRIs and literals can be very long, a dictionary can be used to assign unique IDs to them before applying f_D. This will ensure that the length of the input to $\mathbb{H}$ is bounded, thereby guaranteeing low probability of collision. This will also speed up the computation of $\mathbb{H}$, because the polynomial p in p mod r is now of a much lower degree.

Because $\mathbb{H}$ has very low probability of collision, in practice, we can view $V_{c,SPO}$ as a set, because the quads/triples in a graph are always assumed to be unique. Later in Section 4.2.4, we state a necessary condition for a BGP match using the PV of an RDF graph,

[2] SELECT ?g WHERE { GRAPH ?g { s p o. } }.

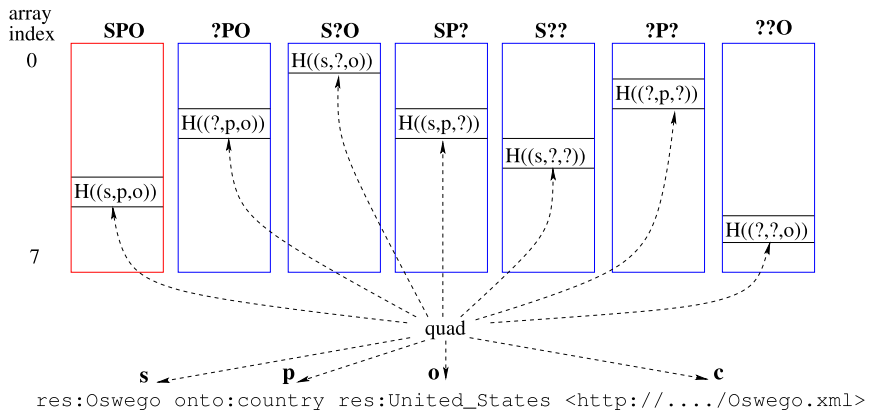


Fig. 5. The PV of G_1. Note that s, p, and o, which appear inside H(·), should be replaced by their actual URIs.

wherein we will perform the subset operation on $V_{c,SPO}$ if the BGP of a query has a triple pattern 's p o' in it. Even if there are collisions, the subset operation on $V_{c,SPO}$ will still succeed, and there will not be any false dismissals. On the other hand, the remaining vectors of $\overline{V_c}$ should be viewed as multisets, because f_D can produce the same output for different triples due to the presence of '?' in the output.

Algorithm 2 PV construction for a BGP

Input: A BGP q
Output: PV $\overline{V_q}$
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** $\overline{V_q}$

Example 2. Let Fig. 5 denote the PV of the RDF graph G_1. For the canonical pattern ?PO, the quad `res:Oswego onto:country res:United_States <http://.../Oswego.xml>` in G_1 is transformed to the tuple `(?,onto:country, res:United_States)` by replacing the subject `res:Oswego` with ?. The hash of the tuple is stored into the vector for ?PO. The figure also shows how the hash values are computed for the other canonical patterns. Once the eight quads of G_1 are processed, each vector of the PV will have eight hash values.

Given a BGP q, we map it into a PV, denoted by $\overline{V_q}$, and compute it slightly differently: Initially, each $V_{q,r}$ is empty. For each triple pattern t in q, we compute $f_Q(t)$ to produce a pair (r, o), where r denotes the canonical pattern for t. We then insert $\mathbb{H}(o)$ into $V_{q,r}$. Algorithm 2 shows the steps involved. As before, $V_{q,SPO}$ can be viewed as a set. The rest of the vectors of $\overline{V_q}$ should be viewed as multisets, because two different triple patterns (each containing at least one variable) in a BGP may hash to the same value. For example, if a BGP contains two triple patterns $?s_1$ `onto` : `utcOffset` $?o_1$ and $?s_2$ `onto` : `utcOffset` $?o_2$, then f_Q('$?s_1$ `onto` : `utcOffset` $?o_1$') = f_Q('$?s_2$ `onto:utcOffset` $?o_2$') and therefore, the hash values produced by $\mathbb{H}$ will be identical.

Note that we will ignore any triple pattern of the type $?v_s$ $?v_p$ $?v_o$ during the PV construction. This is because such a triple pattern will match every triple in an RDF graph. Precisely for this reason, we do not include the canonical pattern ??? in $\mathbb{P}$ as every triple in every RDF graph would produce the same hash value for ???. As a result, maintaining a vector for ??? in the PV of an RDF graph would be wasteful as it does not provide any pruning capability during BGP matching.

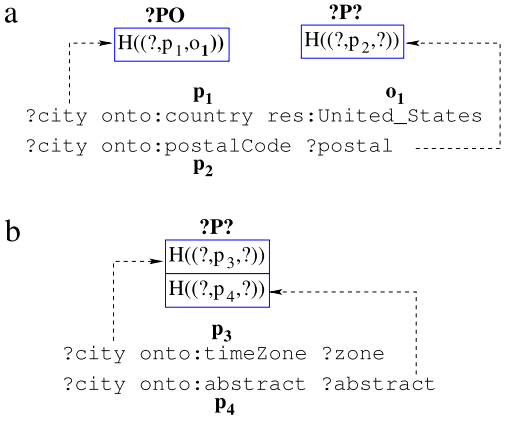


Fig. 6. (a) The PV of BGP_3. (b) The PVs of BGP_1.

Example 3. Consider BGP_3 of query Q. As shown in Fig. 6(a), the triple patterns `?city onto:country res:United_States` and `?city onto:postalCode ?postal` are transformed to tuples `(?,onto:country,res:United_States)` and `(?,onto:postalCode,?)`, respectively. Their hash values are stored in the vectors for ?PO and ?P?, respectively. Fig. 6(b) shows how BGP_1 is mapped into its PV. Because both of its triple patterns produce tuples that match the canonical pattern ?P?, its PV has only one vector.

4.2.3. Operations on pattern vectors

Next, we define two operations on PVs, which will be used during the construction of the filtering index. Our goal is to group similar PVs (and as a result, similar RDF graphs) together so that candidate RDF graphs are identified and processed quickly during query processing.

Definition 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 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 $sim(V_{a,r}, V_{b,r}) = \frac{|V_{a,r} \cap V_{b,r}|}{|V_{a,r} \cup V_{b,r}|}$.

4.2.4. Index construction

We begin by describing a key necessary condition, which forms the basis for indexing and query processing in RIQ. Because we map both the RDF graphs and BGPs into their PVs, we must characterize the relationship between them when processing a BGP via subgraph matching. We state the following theorem.

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

Algorithm 3 The PV-Index Construction

Input: a list of PVs; (k, l, m): LSH parameters; ϵ: false positive rate
Output: filters of all the groups of similar RDF graphs
1: Let $\mathbb{G}(\mathbb{V}, \mathbb{E})$ be initialized to an empty undirected graph
2: **for** each PV $\overline{V_i}$ **do**
3: Add a new vertex v_i to $\mathbb{V}$
4: **for** each $r \in \mathbb{P}$ **do**
5: $\{h_{i1}, ..., h_{ik}\} \leftarrow \mathsf{LSH}_{k,l,m}(V_{i,r})$
6: **for** every $v_j \in \mathbb{V}$ and $i \neq j$ **do**
7: **if** $\exists\, o$ s.t. $1 \leq o \leq k$ and $h_{io} = h_{jo}$ **then**
8: Add an edge $\{v_i, v_j\}$ to $\mathbb{E}$ if not already present
9: Compute the connected components of $\mathbb{G}$. Let $\{C_1, ..., C_t\}$ denote these components.
10: **for** $i = 1$ to t **do**
11: Compute the union U_i of all PVs corresponding to the vertices in C_i
12: Construct a BF for $U_{i,SPO}$ with false positive rate ϵ given the capacity $|U_{i,SPO}|$
13: Construct a CBF for each of the remaining vectors of U_i with false positive rate ϵ given the capacity $|U_{i,*}|$
14: Store the ids of graphs belonging to C_i
15: **return**

Theorem 1. *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 $\bigwedge_{r\in\mathbb{P}}(V_{q,r} \subseteq V_{c,r}) =$ TRUE.*

Proof. 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, $\bigwedge_{r\in\mathbb{P}}(V_{q,r} \subseteq V_{c,r}) =$ TRUE. □

According to Theorem 1, given a BGP, if we can identify those RDF graphs in the database whose PVs satisfy the necessary condition, then we have a superset of RDF graphs that contain a subgraph match for the BGP. This also guarantees that there are no false dismissals.

Example 4. Because BGP_3 in Q has a subgraph match in G_1, the vectors for ?*PO* and ?*P*? in BGP_3's PV (as shown in Fig. 6(a)) are subsets of the vectors for ?*PO* and ?*P*? in G_1's PV, respectively.

Rather than testing every PV in the database – one-at-a-time – during query processing, we propose a novel filtering index called the PV-Index to effectively organize millions of PVs in the database. Using this index, we aim to quickly identify candidate RDF graphs in the early stages of query processing using Theorem 1. Our goal is to discard most of the non-matching RDF graphs without any false dismissals. As a result, the subsequent stages of query processing will process fewer candidates to obtain the final results, thereby speeding up query processing.

There are two issues that arise while designing the PV-Index: First, we want to group similar PVs together (and in an efficient manner) so that for a given BGP, we can quickly discard most of the non-matching RDF graphs. For this, we will employ LSH (Section 2.2.2). Second, we want to compactly store the PV-Index to minimize the cost of I/O during query processing. For this, we will employ BFs and CBFs (Section 2.2.3).

In Algorithm 3, we outline the steps to construct the PV-Index. We build an undirected graph $\mathbb{G}$, where each vertex of $\mathbb{G}$ represents a PV. For every PV, we apply LSH on each of its seven vectors. Suppose there are two PVs such that the application of LSH on their vectors for the same pattern r, produces at least one identical hash value, then we add an edge between the vertices representing these PVs (Lines 2–8). Essentially, a missing edge between two vertices indicates that their corresponding PVs are dissimilar with high probability. Once $\mathbb{G}$ is constructed, we compute (in linear time) the connected components in it. Each connected component represents RDF graphs whose corresponding PVs are similar with high probability. We treat these graphs as a group and compute the union of their PVs (Line 11). The union operation summarizes the PVs as well as preserves the condition stated in Theorem 1. (The individual vectors in a PV are kept sorted so that the union operation can be performed in linear time.)

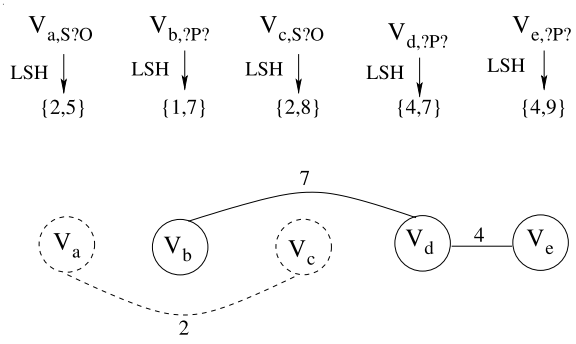


Fig. 7. Grouping five PVs into two connected components ($k = 2$, $m = 10$).

Example 5. Let us consider the example in Fig. 7 with five PVs, V_a, V_b, V_c, V_d, and V_e. Suppose the application of LSH on some of the patterns produces the hash values as shown. Because $V_{a,S?O}$ and $V_{c,S?O}$ share the hash value 2, we add an edge between V_a and V_c in the graph. Also, $V_{b,?P?}$ and $V_{d,?P?}$ share the hash value 7 and, $V_{d,?P?}$ and $V_{e,?P?}$ share the hash value 4. Therefore, we add edges between V_b and V_d and V_d and V_e. Ultimately, we have two connected components.

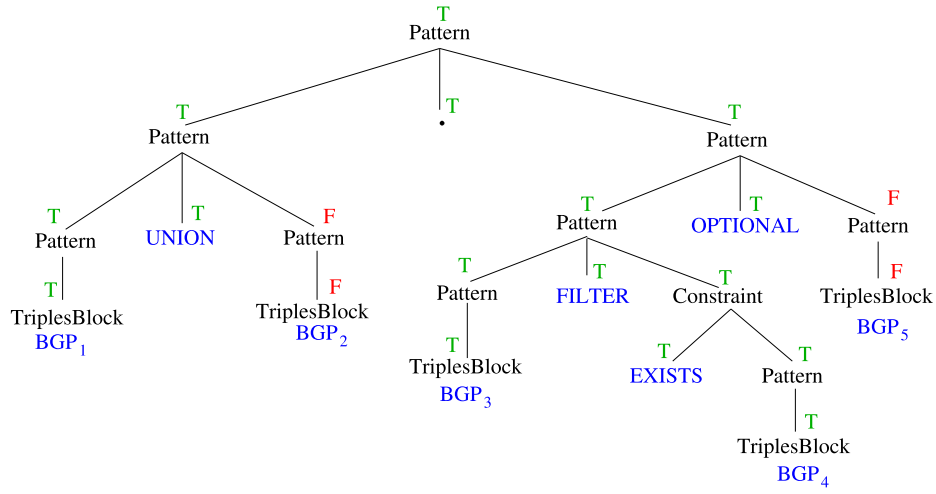
To compactly represent the union computed for a connected component, we use a combination of one Bloom filter (BF) and six Counting Bloom filters (CBFs). The vector for the canonical pattern SPO is stored using a BF and the others are stored using CBFs. Each filter of a vector is configured for a false positive rate of ϵ and capacity equal to the cardinality of the vector (Lines 12 and 13). For each connected component, we also store the IDs of graphs belonging to it. In summary, the BFs and CBFs for all the connected components constitute the PV-Index. Each group of graphs is separately indexed using a tool like Jena TDB.

4.3. Query processing

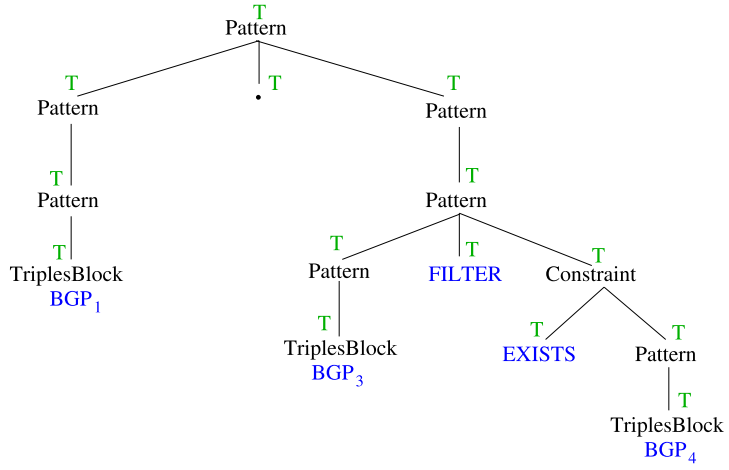
Next, we present the streamlined approach adopted by RIQ for efficient SPARQL query processing via a decrease-and-conquer strategy. RIQ constructs a plan for the query and searches the PV-Index to quickly identify the candidate groups of RDF graphs that may contain a match for the query. It rewrites the original query methodically for each candidate group and executes optimized queries on them (using an existing SPARQL query processor) to produce the final results.

Given a query, the first step is to parse its GRAPH block according to the SPARQL grammar and generate a tree-representation, which we call the BGP Tree. This tree serves as an execution plan for processing individual BGPs in the query. It is evaluated on each connected component to produce an optimized query, which is then executed on that connected component. As an example, consider the query in Fig. 2, which is represented by the BGP Tree in Fig. 8(a).

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005



(a) A processed BGP Tree.



(b) Pruned BGP Tree.

Fig. 8. Query processing.

Given a node n in the BGP Tree, let $eval[n]$ be a Boolean variable for the node to capture the status of the evaluation of the BGP Tree on a connected component of the PV-Index. We initialize $eval[n] =$ `FALSE` for every node in the tree. We invoke Algorithm 4 on each connected component, starting from the root of the BGP Tree in depth-first order. When a node represents a BGP (Line 1), we test the necessary condition stated in Theorem 1 by calling Algorithm 5. This involves the processing of membership queries on the BF and CBFs constructed for that connected component. When a node represents the `AND` operation (*i.e.*, '.'), and a child of that node evaluates to `FALSE`, we skip processing the other child node (Line 6). This is because the RDF graphs belonging to that connected component will not produce a match for the subexpression rooted at that node. When a node represents a `UNION` operation, at least one of the children should evaluate to `TRUE` to produce a match (Line 14). While the node under `OPTIONAL` is evaluated (Line 17), the status of the evaluation is ignored during the BGP Tree evaluation (Line 18) because of the semantics of `OPTIONAL` in SPARQL. However, the status of the node will be used during the generation of an optimized query.

If $eval[root] =$ `TRUE`, then the group of RDF graphs belonging to that connected component is a candidate for further processing. For this candidate, an optimized SPARQL query can be generated by traversing the BGP Tree and checking the evaluation status of each node. We prune the BGP Tree to produce a pruned BGP Tree

```
SELECT ?g ?city ?area ?code ?zone ?abstract ?postal ?popu
WHERE {
  GRAPH ?g {
    { ?city onto:areaLand ?area .
      ?city onto:areaCode ?code . }
    ?city onto:country res:United_States .
    ?city onto:postalCode ?postal .
    FILTER EXISTS { ?city onto:utcOffset ?offset . }
  }}
```

Fig. 9. Optimized query.

from which the optimized query can be constructed. Algorithm 6 shows the steps involved during the pruning process starting from the root of the BGP Tree. Any predicates within `FILTER` will be included in the optimized query. All the projected variables in the original query are projected in the optimized query. *It should be noted that the optimized query is executed on the candidate for which the BGP Tree was evaluated.* An optimized query is executed on a candidate using an existing tool like Jena TDB or Virtuoso. The results from all the candidates are combined to produce the final output of the query.

Fig. 8(b) shows the pruned BGP Tree for the example in Fig. 8(a). Based on pruned BGP Tree, we generate the optimized query shown in Fig. 9. In this query, the `OPTIONAL` block and one block in the `UNION` are absent.

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

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

if n represents `TriplesBlock` **then**
  Let q denote the BGP
  $eval[n] \leftarrow$ IsMatch(q, j)
else if n represents n_1 . n_2 **then**
  $eval[n_1] \leftarrow$ EvalBGPTree(n_1, j)
  if $eval[n_1]$ is `FALSE` **then**
    $eval[n] \leftarrow$ `FALSE` {//skip processing n_2}
  else
    $eval[n_2] \leftarrow$ EvalBGPTree(n_2, j)
    $eval[n] \leftarrow eval[n_1] \wedge eval[n_2]$
else if n represents n_1 `UNION` n_2 **then**
  $eval[n_1] \leftarrow$ EvalBGPTree(n_1, j)
  $eval[n_2] \leftarrow$ EvalBGPTree(n_2, j)
  $eval[n] \leftarrow eval[n_1] \vee eval[n_2]$
else if n represents n_1 `OPTIONAL` n_2 **then**
  $eval[n_1] \leftarrow$ EvalBGPTree(n_1, j)
  $eval[n_2] \leftarrow$ EvalBGPTree(n_2, j)
  $eval[n] \leftarrow eval[n_1]$
else if n represents n_1 `FILTER` n_2 **then**
  $eval[n_1] \leftarrow$ EvalBGPTree(n_1, j)
  $eval[n_2] \leftarrow$ EvalBGPTree(n_2, j)
  $eval[n] \leftarrow eval[n_1] \wedge eval[n_2]$
else if n represents `EXISTS` n_1 **then**
  $eval[n_1] \leftarrow$ EvalBGPTree(n_1, j)
  $eval[n] \leftarrow eval[n_1]$
else if n represents `NOT EXISTS` n_1 **then**
  $eval[n_1] \leftarrow$ EvalBGPTree(n_1, j)
  $eval[n] \leftarrow$ `TRUE`
else if n represents `Expression` **then**
  $eval[n] \leftarrow$ `TRUE` {//skip processing predicates}
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
for each $r \in \mathbb{P}$ **do**
  Construct $\mathbb{F}_{q,r}$ with the same capacity and false positive rate as $\mathbb{F}_{U_j,r}$
for each bit in $\mathbb{F}_{q,SPO}$ set to 1 **do**
  if the corresponding bit in $\mathbb{F}_{U_j,SPO}$ is 0 **then**
    return `FALSE`
for each $r \in \mathbb{P} \setminus \{SPO\}$ **do**
  for each non-zero counter in $\mathbb{F}_{q,r}$ **do**
    Let c be the counter value
    if the corresponding counter in $\mathbb{F}_{U_j,r}$ is less than c **then**
      return `FALSE`
return `TRUE`

Algorithm 6 PruneBGPTree(node n)

if $eval[n] =$ `FALSE` **then**
  if n is a `Pattern` of `NOT EXISTS` **then**
    return
  else if n is a `Pattern` of `UNION` or `OPTIONAL` **then**
    Prune away `UNION` or `OPTIONAL`
  Prune away the subtree rooted at n
else
  for each child c of n **do**
    PruneBGPTree(c)
return

5. Performance evaluation

In this section, we report a comprehensive performance evaluation of RIQ on a real and a synthetic dataset, each with about 1.4 billion RDF statements.

5.1. Software, implementation, and hardware setup

We compared RIQ with state-of-the-art RDF engines designed to support named graph queries: Jena TDB and Virtuoso. Both of them can index RDF datasets with billions of quads and process SPARQL queries with the `GRAPH` keyword. Note that Virtuoso is a commercial tool and has been heavily optimized over the years. We also compared RIQ with a popular approach for indexing and query processing RDF triples, namely, RDF-3X. However, for a fair comparison, we employed a reification approach to map quads to triples before indexing with RDF-3X. We also transformed the original SPARQL queries (with the `GRAPH` keyword) so that RDF-3X could produce the correct results. We provide more details on the reification approach in Section 5.5

In our experiments, we used Apache Jena 2.11.1 (TDB), Virtuoso Open-Source Edition 7.1.0, and the latest version of RDF-3X. RDF-3X and Virtuoso are written in C++. Jena TBD is a Java codebase. RIQ is a C++ codebase and uses popular open-source libraries for parsing RDF data [83] and constructing BFs and CBFs [84]. We ran all the experiments on a 64-bit Ubuntu 12.04 machine with 4 Intel Xeon 2.4 GHz cores and 16 GB RAM.

5.2. Datasets and queries

We used a real and a synthetic dataset in our experiments. The real dataset was BTC 2012 [11], which is widely used in the Semantic Web community. It contained 1.36 billion RDF quads with 57,000 unique predicates and 9.59 million RDF named graphs. As we did not readily have access to a synthetic RDF dataset with quads, we decided to generate one using the Lehigh University Benchmark (LUBM) [85]. We generated a dataset with 1.38 billion triples, 18 unique predicates, and 10,000 universities. The triples were divided across 200,004 files, and each file was treated as an RDF graph. Thus, we assumed the context for each triple to be a URI based on the file name. Our sole purpose of doing so was to test RIQ's performance for processing named graph queries on a well-known synthetic dataset. In all, LUBM had 200,004 RDF named graphs. One should note that while BTC 2012 contained RDF data crawled from the Web and had a large number of distinct properties, LUBM was very homogeneous and had a small number of unique predicates.

We used a query workload with varying number of named graph matches to test the performance impact of RIQ's *decrease-and-conquer* approach. In Table 2, we list the queries for both BTC 2012 and LUBM. The table also shows the number of BGPs and triples patterns in each query along with the number of results and matching named graphs for that query. Within each category (i.e., large, small, or multiple BGPs), the queries are ordered by the increasing number of named graph matches. The actual queries are listed in Appendix A.

For BTC 2012, the query set included 2 SPARQL queries with large, complex BGPs (B1–B2) and 5 others (B3–B7) with small BGPs. In addition, there were 4 queries (B8–B11) with multiple BGPs combined using constructs like `UNION` and `OPTIONAL`. Note that B10 and B11 were derived from the DBpedia SPARQL Benchmark [86]. Overall, these queries matched a small fraction of the named graphs in the dataset and had high selectivity. (BTC 2012 had a total of 9.59 million RDF named graphs.)

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

Table 2
Queries for BTC 2012 and LUBM. Note (l) indicates a query has a large BGP, (s) indicates a query has a small BGP, and (m) indicates a query has multiple BGPs. Within each group, the queries are ordered by the number of named graph matches.

Dataset	Query	# of BGPs	# of triple patterns	# of results/# of named graphs matched
	B1 (l)	1	19	6/1
	B2 (l)	1	21	5/2
B	B3 (s)	1	5	0/0
T	B4 (s)	1	5	12,101,709/2020
C	B5 (s)	1	6	146,012/3691
	B6 (s)	1	7	1,460,748/3691
2	B7 (s)	1	4	47,493/6413
0	B8 (m)	7	12	196/1
1	B9 (m)	4	7	149,306/25,016
2	B10 (m)	5	8	249,318/25,016
	B11 (m)	2	5	525,435/123,171
	L1 (l)	1	22	0/0
	L2 (l)	1	18	24/1
	L3 (l)	1	23	16/6
	L4 (s)	1	6	0/0
L	L5 (s)	1	4	172/21
U	L6 (s)	1	5	8341/21
B	L7 (s)	1	6	440,834/175,559
M	L8 (s)	1	6	468,047/179,847
	L9 (s)	1	2	10,798,091/200,004
	L10 (s)	1	1	25,205,352/200,004
	L11 (s)	1	1	79,163,972/200,004

For LUBM, the query set included 3 SPARQL queries with large, complex BGPs (L1–L3) and 8 others (L4–L11) with small BGPs that are variations of the queries in the LUBM benchmark. Queries L1–L6 matched a small fraction of the named graphs in the dataset and had high selectivity. On the other hand, queries L7–L8 matched about 88% of the named graphs in LUBM, and L9–L11 matched 100% of the named graphs in the dataset. We regard L7–L11 to be low selectivity queries. (LUBM had a total of 200,004 RDF named graphs.) Note that each query in this query set had only one BGP.

One may wonder if blank nodes are supported by RIQ as the queries used in our experiments do not contain blank nodes. In SPARQL [10], blank nodes in a query act as variables. So it is possible to support blank nodes in RIQ by replacing each distinct blank node in a BGP with a unique variable name. The PV of the BGP can be generated as before using Algorithm 2. The newly added variables will not be projected in the query.

5.3. Evaluation metrics

For comparing the query processing performance of RIQ and its competitors, we measured the wall-clock time taken by each approach to process a query in the cold and warm cache settings, and computed the average over 3 runs. We dropped the file system buffer cache by issuing the command `echo 3 > /proc/sys/vm/drop_caches`. For RIQ, we also measured the filtering time for a query (using the PV-Index) along with the % of total query processing time consumed by the filtering phase, the number of candidate groups identified by the filtering phase, and the number of candidate groups containing true matches for the query.

For the experiments, Jena TDB was executed with its default statistics-based optimization. Virtuoso was executed with its default settings and optimizations enabled. Whenever an approach ran for more than four hours, we report the time taken as "14400+".

5.4. SPARQL processor used by RIQ

RIQ relies on an existing SPARQL processor that supports quads (*e.g.*, Jena TDB, Virtuoso) to query the candidate groups obtained after filtering. In our experiments, RIQ used either Virtuoso or Jena TDB depending on the number of candidate groups identified by the filtering phase. RIQ used Virtuoso whenever the filtering phase identified just a couple of candidate groups for a query and Jena TDB otherwise. This is because Virtuoso is a client–server software, where the Virtuoso server performs the indexing and query processing on a given database. The Virtuoso client accepts queries and sends them to the Virtuoso server for processing. So when RIQ used Virtuoso, we had to ensure that all the Virtuoso servers for the candidate groups were running and ready to process the optimized queries generated by RIQ immediately after the filtering phase. It took more than 60 s to start a Virtuoso server on a candidate group/database. Thus, when the filtering phase identified a large number of candidate groups (e.g., 10 or more) for a query, it became tedious to conduct the experiments and also added unnecessary load on our machine. In such a case, RIQ programmatically invoked Jena TDB as the underlying SPARQL processor.

In the context of the queries shown in Table 2, note that RIQ used Virtuoso as the underlying SPARQL processor for queries B1, B2, L1, and L2. For the remaining queries, RIQ used Jena TDB. The number of candidates groups identified for each query is presented later in Tables 5 and 7.

5.5. Performance evaluation on queries with a single BGP

We conducted the first set of experiments to compare RIQ with its competitors for queries with a single BGP (*i.e.*, B1–B7 and L1–L11). Our goal was to demonstrate the effectiveness of RIQ's PV-Index and *decrease-and-conquer* strategy for efficient query processing.

5.5.1. Approaches compared

RIQ, Jena TDB, and Virtuoso built their respective indexes on quads in the datasets. Because RDF-3X is designed to query triples, for fair comparison, we employed a reification approach to represent quads in a dataset as triples. We computed a unique ID `<t>` for each quad `<s> <p> <o> <c>` and represented that quad as a set of four triples, namely, `<t> <first> <s>`, `<t> <second> <p>`, `<t> <third> <o>`, and `<t>`

Table 3
RIQ's index construction cost.

Dataset	Construction time (in s)				LSH	# of	Avg. # of	False	Max.	PV-Index
	PVs	Connected components	BF/CBFs	Total	params. (k, l)	unions	graphs per union	+ve rate (ϵ)	filter capacity (M)	size (GB)
BTC 2012	16,700	27,348	2476	46,524	(4,6)	526	18,232	5%	1	6.5
LUBM	15,249	22,711	3402	41,362	(30,4)	487	411	1%	10	12

Table 4
Query processing performance on BTC 2012. The winning approach is shown in bold within shaded cells.

Query	Cold cache Time taken (in s)					Warm cache Time taken (in s)				
	RIQ	RIQ$^-$	RF	Jena TDB	Virtuoso	RIQ	RIQ$^-$	RF	Jena TDB	Virtuoso
B1 (l)	**4.06**	148.85	14400+	15.79	5.92	0.67	100.71	14400+	13.15	**0.13**
B2 (l)	**3.73**	158.14	14400+	23.21	6.04	**1.53**	114.38	14400+	20.51	4.43
B3 (s)	22.85	157.49	296.99	16.58	**4.50**	15.06	101.93	276.72	13.30	**1.43**
B4 (s)	**293.42**	412.61	1010.77	295.77	965	256.22	398.42	990.72	**148.39**	950.07
B5 (s)	**65.40**	183.16	14400+	668.28	86.71	43.18	104.38	14400+	21.65	**10.33**
B6 (s)	**102.96**	219.10	14400+	684.97	350.19	76.90	138.85	14400+	**56.59**	342.79
B7 (s)	**66.35**	183.08	405.29	803.94	81.94	41.18	103.29	387.50	17.09	**2.69**
B8 (m)	**16.29**	796.54	n/a	3564.44	39.18	6.79	493.76	n/a	369.81	**0.16**
B9 (m)	**110.70**	227.37	n/a	648.93	142.89	57.82	117.01	n/a	**33.36**	60.42
B10 (m)	**116.74**	232.37	n/a	663.31	165.52	62.61	123.19	n/a	**38.77**	88.36
B11 (m)	**158.18**	272.0	n/a	2052.62	237.58	**76.68**	133.42	n/a	2102.06	120.28

<fourth> <c>. We converted a SPARQL query with the GRAPH keyword (on quads) into an equivalent SPARQL query on the reified version of the dataset. Note that this transformation can be done in linear time. For example, the query.

```
SELECT ?u ?v ?w WHERE {
 GRAPH ?g { ?u <p> ?v . ?u <q> ?w .}}
```

was transformed into

```
SELECT ?u ?v ?w WHERE {
  ?t0 <first> ?u . ?t0 <second> <p> .
  ?t0 <third> ?v . ?t0 <fourth> ?g .
  ?t1 <first> ?u . ?t1 <second> <q> .
  ?t1 <third> ?w . ?t1 <fourth> ?g . }.
```

Hereinafter, we will refer to RDF-3X operating on the reified version of a dataset as RF.

To understand how much performance improvement RIQ achieves using the PV-Index, we measured the total time taken to execute a query without the filtering phase in RIQ. That is, we assumed all the connected components of the filtering index were candidate groups for an input query and executed the query on all the groups. (Each candidate group is indexed using Jena TDB or Virtuoso.) We will refer to this approach as RIQ$^-$.

5.5.2. Index construction

The size of BTC 2012 and LUBM was 218 GB and 217 GB, respectively. Jena TDB indexed BTC 2012 in 139,804 s, and the index size was 275 GB. It indexed LUBM in three and a half days, and the index size was 280 GB. Virtuoso indexed BTC 2012 in five and a half days, and the index size was 77 GB. It indexed LUBM in 3 days, and the index size was 43 GB. RF indexed the reified version of BTC 2012 and LUBM in about 3 days each. The index size for BTC 2012 and LUBM was 274 GB and 259 GB, respectively.

Next, we report the filtering index construction cost of RIQ on BTC 2012 and LUBM. Table 3 shows the breakdown of the total PV-Index construction time including the time to construct the PVs, the connected components, and the BF/CBFs. During the construction of PVs, we first assigned unique IDs to URIs and literals in a dataset before applying the hash function $\mathbb{H}$, which produced 32-bit hash values as output. The LSH parameters used for the PV-Index construction are also shown in the table. Since LUBM was more homogeneous than BTC 2012, we used a lower similarity cutoff for LUBM than BTC 2012 to create a reasonable number of groups with similar RDF graphs. The PV-Index for BTC 2012 and LUBM had 526 and 487 connected components, respectively. The average number of graphs per connected component, the parameters used to tune the filters, and the size of the PV-Index are also reported in Table 3. Overall, the size of the PV-Index was less than 6% of the total dataset size. This shows that the PV-Index index is indeed compact, which can facilitate fast pruning of the groups of similar graphs during query processing.

5.5.3. Processing queries with a large, complex BGP

In this subsection, we report the performance evaluation results of RIQ and its competitors for processing named graph queries containing a large, complex BGP on RDF quads. Each large, complex BGP had at least one undirected cycle.

BTC 2012. The results for B1 and B2 on BTC 2012, a dataset with 9.59 million RDF named graphs, are reported in Table 4. In the cold cache setting, RIQ outperformed its competitors for both B1 and B2, each of which had a large, complex BGP in it. While RIQ was more than 3 times faster than Jena TDB for both the queries, it was also able to beat Virtuoso, a commercial tool. In Table 5, we report the filtering performance of RIQ for B1 and B2. Both B1 and B2 were high selectivity queries with matches in one and two named graphs, respectively. RIQ filtering index was effective in quickly identifying a small number of candidate groups from a total of 526 groups/unions followed by efficient refinement on the candidate groups. The average number of named graphs per group was 18,232 as reported in Table 3. Therefore, it was faster than both

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

Table 5
RIQ's filtering cost on BTC 2012.

Query	Filtering cost Filtering time (in s), % of total time		# of candidate groups identified/processed	# of groups w/ matches
	Cold cache	Warm cache		
B1	1.53, 37.68%	0.18, 26.87%	1	1
B2	1.28, 34.32%	0.13, 8.50%	2	2
B3	3.86, 16.89%	0.15, 1.00%	63	0
B4	1.67, 0.57%	0.08, 0.03%	140	112
B5	1.67, 37.68%	0.09, 0.23%	166	73
B6	1.78, 1.73%	0.10, 0.13%	164	73
B7	1.75, 2.64%	0.09, 0.23%	173	102
B8	6.42, 39.41%	0.66, 9.72%	18	1
B9	5.12, 4.63%	0.78, 1.35%	195	145
B10	5.15, 4.41%	0.85, 1.36%	195	145
B11	6.21, 3.60%	0.61, 0.80%	243	221

Table 6
Query processing performance on LUBM. The winning approach is shown in bold within shaded cells. A * indicates that the approach aborted after the reported time.

Query	Cold cache Time taken (in s)					Warm cache Time taken (in s)				
	RIQ	RIQ$^-$	RF	Jena TDB	Virtuoso	RIQ	RIQ$^-$	RF	Jena TDB	Virtuoso
L1 (l)	**10.63**	66.94	14400+	233.59	136.62	**2.15**	45.17	14400+	2.76	77.99
L2 (l)	**14.40**	214.70	14400+	520.91	55.87	3.13	47.80	14400+	5.05	**0.30**
L3 (l)	**64.95**	211.05	14400+	523.78	119.62	**9.61**	207.12	14400+	244.82	36.58
L4 (s)	5.24	87.49	14400+	**4.83**	13.96	0.23	46.64	14400+	1.10	**0.003**
L5 (s)	14.37	79.09	511.88	**6.16**	8.925	2.93	45.21	486.48	1.43	**0.005**
L6 (s)	15.52	79.30	600*	**9.19**	15.315	5.08	46.66	600*	3.67	**0.16**
L7 (s)	755.35	–	14400+	2349.70	**134.50**	741.32	–	14400+	2346.31	**43.04**
L8 (s)	759.17	–	14400+	2357.70	**140.46**	733.81	–	14400+	2353.21	**37.43**
L9 (s)	549.55	–	1232.02	1432.42	**166.35**	533.52	–	1219.84	1445.41	**137.42**
L10 (s)	482.12	–	1327.42	14400+	**320.67**	499.20	–	1315.40	14400+	**303.62**
L11 (s)	**814.62**	–	1521.32	1511*	961.22	**805.20**	–	1485	1511*	929.37

Table 7
RIQ's filtering cost on LUBM.

Query	Filtering cost Filtering time (in s), % of total time		# of candidate groups identified/processed	# of groups w/ matches
	Cold cache	Warm cache		
L1	4.72, 44.40%	0.20, 9.11%	2	0
L2	4.36, 30.27%	0.18, 5.75%	1	1
L3	6.12, 9.42%	0.21, 2.19%	14	6
L4	5.39, 100.00%	0.23, 100.00%	0	0
L5	8.66, 48.06%	0.19, 6.79%	17	17
L6	7.26, 36.37%	0.20, 4.06%	17	17
L7	9.71, 1.29%	0.20, 1.30%	487	437
L8	9.27, 1.22%	0.21, 1.27%	487	453
L9	8.17, 1.39%	0.16, 1.45%	487	487
L10	6.07, 1.26%	0.15, 0.02%	487	487
L11	5.96, 0.73%	0.10, 0.74%	487	487

Jena TDB and Virtuoso. RF was the slowest because the number of RDF statements was four times more than the original dataset and the transformed SPARQL queries on the reified version of the dataset contained very large number of triple patterns (*i.e.*, 72 triple patterns for B1 and 80 triple patterns for B2). This increased the number of join operations and significantly slowed down the query processing in RF.

The results for the warm cache setting are shown in Table 4. RIQ was significantly faster than Jena TDB and RF. While RIQ was almost three times faster than Virtuoso for B2, it was slower than Virtuoso for B1.

We also compared RIQ with RIQ$^-$ to gauge the benefit provided by the filtering index during query processing. Clearly, as shown in Table 4, RIQ was significantly faster than RIQ$^-$ for B1 and B2 as it had to process at most 2 candidate groups instead of all the groups.

LUBM. The results for LUBM, a dataset with 200,004 RDF named graphs, are reported in Table 6. In the cold cache setting, RIQ processed queries with large, complex BGPs (L1–L3) significantly faster than Virtuoso, Jena TDB, and RF. For example, RIQ processed L2 almost 4 times faster than Virtuoso. It was about twice as fast

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

as Virtuoso for L3, where six named graphs were matched for the query. We report RIQ's filtering cost for each query along with the number of candidate groups processed in Table 7. As L1–L3 had high selectivity, RIQ's decrease-and-conquer approach was highly effective is pruning a large portion of the dataset to quickly identify the candidate groups containing matches. As reported in Table 3, the total number of groups/unions was 487 and the average number of named graphs per group was 411. As before, RF was the slowest among all the approaches due to the large number of triple patterns in the transformed SPARQL queries due to the reification approach. The results for the warm cache setting are shown in Table 6. For queries L1 and L3, RIQ was the fastest. But for L2, Virtuoso was the fastest. Jena TDB and RF were slower than RIQ and Virtuoso.

Once again, RIQ$^-$ was consistently slower than RIQ, which attests the benefit of RIQ's filtering index for the tested queries.

From the above experiments, we conclude that RIQ's *decrease-and-conquer* approach yields superior performance over to its competitors for named graph queries containing large, complex BGPs, when I/O is the dominating factor in the total query processing time. Such queries tend to have high selectivities, and therefore, RIQ filtering index is effective in pruning away a large portion of the named graphs in a dataset that did not contain matches. In the warm cache setting, RIQ achieved good performance overall by winning on three of the five queries.

5.5.4. Processing queries with a small BGP

In this subsection, we present the performance evaluation results for processing named graph queries containing a small BGP.

BTC 2012. The tested queries (B3–B7) contained less than 8 triple patterns. They were different in the number of results output and the number of matching named graphs. This was a good query workload to evaluate how the performance of RIQ was affected by an increase in the number of matching named graphs in the dataset.

As shown in Table 4, RIQ was faster than its competitors in the cold cache setting for four out of the five queries. For example, RIQ processed B6 three times faster than Virtuoso. Virtuoso was faster than RIQ only for B3, where the query returned no results. RIQ identified 63 candidate groups although none of them had true matches.

RIQ's filtering cost for B3–B7 along with the number of candidate groups processed are reported in Table 5. Compared to queries B1–B2, which contained large, complex BGPs, these queries (expect B3) returned more results and had higher number of named graph matches. As a result, the number of candidate groups identified after the filtering phase was much higher than that for B1–B2. Despite this, RIQ was able to outperform its competitors except for B3. This shows that RIQ's filtering index effectively prunes away a large portion of the RDF named graphs in the dataset and its decrease-and-conquer approach is suitable for a variety of SPARQL queries on named graphs in the cold-cache setting. Jena TDB was slower than RIQ and Virtuoso in most cases.

In the warm cache setting, Virtuoso was the fastest for three out of the five queries, and Jena TDB was the fastest in two of them. The filtering time of RIQ in this setting is shown in Table 5.

In both the cold and warm cache settings, RIQ consistently outperformed RIQ$^-$ for queries B3–B7 due to the use of the filtering index.

LUBM. Recall that compared to BTC 2012, LUBM had a small number of unique predicates. Table 6 shows the results for LUBM queries L4–L6 in the cold cache setting. RIQ was comparable in performance with Virtuoso as neither was a clear winner. Jena TDB, on the other hand, was faster than RIQ and Virtuoso. Overall, RIQ's filtering index effectively pruned away a large number of the named graphs in the dataset that did not contain matches for a query. (See Table 7.) However, on L7–L11, which matched a large portion of the named graphs in the dataset (above 88%), Virtuoso was the fastest in four out of the five queries. While RIQ was superior in performance to Jena TDB and RF, RIQ's decrease-and-conquer approach was less effective as compared to Virtuoso. This is because the filtering index provided no pruning power as almost all the named graphs in the dataset had matches. As shown in Table 7, for L7–L11, 487 candidate groups were identified by the filtering phase of RIQ. Table 6 shows the results for LUBM in the warm cache setting. Except for L11, Virtuoso was clearly faster than other techniques. (Note that Jena TDB and RF aborted during the execution of queries L11 and L6, respectively.)

We compared RIQ with RIQ$^-$ on L4–L6 because the filtering index was effective in pruning away a large portion of the named graphs in the dataset. RIQ was significantly faster than RIQ$^-$, and this attests the benefit of the filtering index for such queries.

From the above results, we conclude RIQ's strategy of query processing yields superior performance on high selectivity queries that matched a small fraction of the named graphs in the dataset, when the I/O cost was the dominating factor *i.e.*, in the cold cache setting. However, Virtuoso was more efficient than RIQ on queries that matched most of the named graphs in the dataset.

Table 8
Geometric mean of the query processing times for BTC 2012. The winning approach is shown in bold within shaded cells.

Setup	Query	Geo. mean (in s)			
		RIQ	RF	Jena (TDB)	Virtuoso
Cold	B1–B2	**3.8**	14,400	19.1	5.9
	B3–B7	**78.6**	1,907	282.6	100.1
	B1–B7	**33.3**	3,398	130.9	44.7
Warm	B1–B2	1.0	14,400	16.4	**0.7**
	B3–B7	55.5	1,856	33.3	**26.2**
	B1–B7	17.7	3,323	27.2	**9.5**

5.5.5. Comparison based on geometric mean

Finally, we compared RIQ, RF, Virtuoso, and Jena TDB by computing the geometric mean of wall-clock time for the queries. Table 8 shows the geometric mean for queries on BTC 2012 by considering B1–B2, B3–B7, and all the queries B1–B7. For the cold cache setting, RIQ was the winner in all cases. For the warm cache setting, Virtuoso was the winner in all cases. Table 9 shows the geometric mean for queries on LUBM by considering L1–L3, L4–L11, and all the queries L1–L11. RIQ was the winner for L1–L3 in both cold and warm cache settings. However, Virtuoso was the winner for L4–L11 and L1–L11 in both cold and warm cache settings.

5.6. Performance evaluation on queries with multiple BGPs

We conducted the next set of experiments to compare the performance of RIQ with its competitors on queries with multiple BGPs combined using constructs like `UNION` and `OPTIONAL`. Our goal was to demonstrate the effectiveness of RIQ's streamlined approach for efficient query processing. Only Jena TDB and Virtuoso were considered as competitors as they support queries with such constructs. They were run using their default settings and optimizations enabled. We also measured the time taken by RIQ$^-$, wherein the filtering index was not used, and the query was executed on all the groups.

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

Table 9
Geometric mean of the query processing times for LUBM. The winning approach is shown in bold within shaded cells.

Setup	Query	Geo. mean (in s)			
		RIQ	RF	Jena (TDB)	Virtuoso
Cold	L1–L3	**21.5**	14,400	399.4	97.02
	L4–L11	154.4	3,246	425.0	**80.9**
	L1–L11	90.2	5,580	416.3	**85.0**
Warm	L1–L3	**4.0**	14,400	13.7	9.3
	L4–L11	70.99	3,202	290.83	**12.5**
	L1–L11	32.4	5,533	104.9	**11.8**

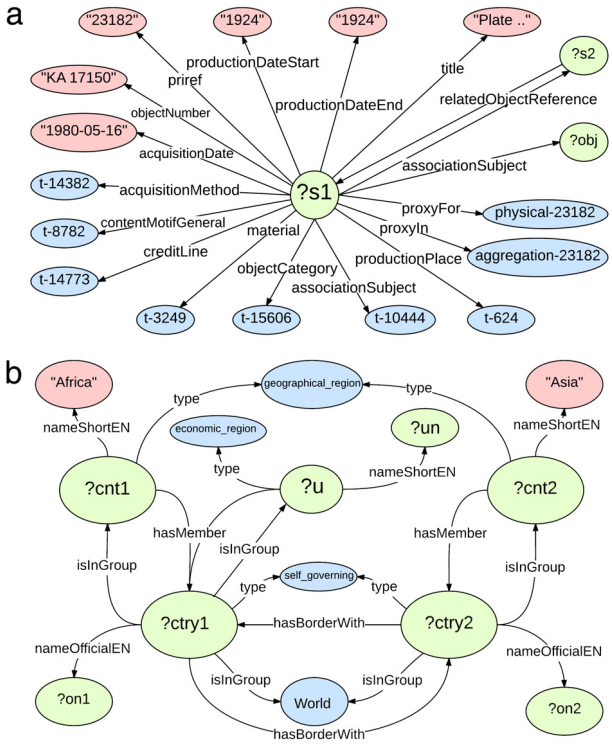


Fig. B.10. Visual representation of BTC queries with large, complex BGPs.

5.6.1. Query processing

The nature of the tested queries is reported in Table 2. Note that the number of matching named graphs matched for queries B8–B11 was a small fraction of the total number of named graphs in BTC 2012. As before, we measured the wall-clock time taken to process B8–B11 in both cold and warm cache settings, and report the average over 3 runs. Table 4 shows the results for the cold cache setting. RIQ outperformed both Jena TDB and Virtuoso for all the four queries. RIQ's filtering index was effective in pruning away a large portion of the named graphs in the dataset that did not contain any matches. On B8, RIQ showed the best improvement over Virtuoso and was about 2.4 times faster, where RIQ identified a total of 18 candidate groups out of 526 after the filtering phase. (See Table 5 for the number of candidate groups and number of groups with matches for queries B8–B11.) On the other queries, RIQ was 1.3–1.5 times faster than Virtuoso. Jena TDB was the slowest of the three approaches in the cold cache setting. For example, Jena TDB was about 6 times slower than RIQ. Table 4 also shows the results for the warm cache setting. RIQ was the fastest for B11; Jena TDB was the fastest for B9 and B10; and

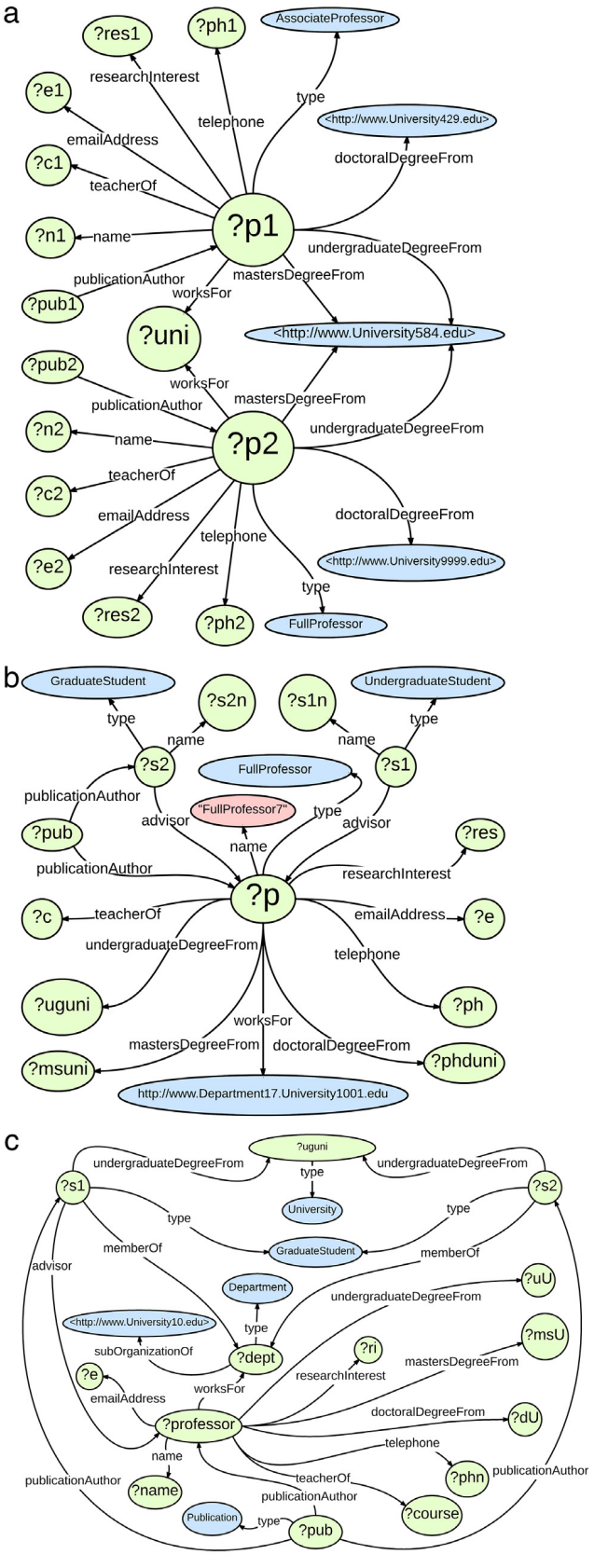


Fig. B.11. Visual representation of LUBM queries with large, complex BGPs.

Virtuoso was the winner for B8. To summarize, RIQ's decrease-and-conquer approach yielded superior performance especially when

Table 10
Geometric mean for queries B8–B11 on BTC 2012. Best results are shown in bold within shaded cells.

Cold cache Time taken (in s)			Warm cache Time taken (in s)		
RIQ	**Jena TDB**	**Virtuoso**	RIQ	**Jena TDB**	**Virtuoso**
76.1	1331.8	121.8	37.2	178.0	**17.9**

I/O was the dominating factor during query processing, *i.e.*, cold cache setting.

For B8–B11, RIQ's filtering time, percentage of total query processing time spent during filtering, the number of candidate groups matched after the filtering phase, and the number of groups with true matches are shown in Table 5. As the number of candidate groups increased across B8–B11 due to the increase in the number of named graph matches, the filtering time did not vary substantially, but the refinement cost increased as expected leading to an increase in the total query processing time (Table 4). Since each query (i.e., B8–B11) had more than one BGP, the total filtering time was usually higher than for that for B3–B7. For each query, we also measured the time taken by RIQ for parsing, BGP Tree evaluation, and query rewriting. This cost was under 0.07 s for each query.

Finally, we measured the performance improvement gained by RIQ using the filtering index. In both the cold and warm cache setting, RIQ was faster than RIQ$^-$ as the filtering phase identified a subset of the groups as candidates for the refinement phase unlike RIQ$^-$ which processed all the groups. The timing results are reported in Table 4, and the number of candidate groups identified for each query during the filtering phase is reported in Table 5.

The geometric means are shown in Table 10. As before, RIQ was the winner in the cold cache setting, and Virtuoso was the winner in the warm cache setting.

5.7. Summary of results

Below we summarize the key findings of our performance evaluation.

- On SPARQL queries with large, complex BGPs for BTC 2012 and LUBM, RIQ outperformed its competitors consistently when I/O was the dominating factor, *i.e.*, in the cold cache setting. Each of these queries had high selectivity with a small number of named graph matches. The filtering index of RIQ provided significant benefit during query processing via its *decrease-and-conquer* approach by pruning away a large portion of the named graphs in the dataset that did not contain matches for the query. As a result, RIQ's query processing time was considerably lower than its competitors. In the warm cache setting, RIQ achieved good performance overall by being the fastest for three out of the five queries.
- On SPARQL queries with small BGPs for BTC 2012 – a real dataset with large number of distinct properties and named graphs – RIQ was the fastest in the cold cache setting for four out of the five queries. Once again, the filtering index of RIQ was very effective for these high selectivity queries by pruning away a large portion of the named graphs in the dataset that did not contain matches. In the warm cache setting, Virtuoso and Jena TDB performed better than RIQ.
- On the contrary, for LUBM – a synthetic dataset with small number of predicates and quite homogeneous in nature – RIQ was comparable in performance with Virtuoso when the queries matched a small fraction of named graphs in the dataset. But for low selectivity queries on LUBM that matched most of the named graphs in the dataset, Virtuoso was the fastest, and its indexing and query processing approach was superior to RIQ's decrease-and-conquer approach.
- On SPARQL queries with multiple BGPs combined using constructs like `UNION` and `OPTIONAL` on BTC 2012, RIQ outperformed its competitors when I/O was the dominating factor. These queries matched a small number of named graphs in the dataset. RIQ's streamlined approach to processing a query starting with BGP Tree construction and evaluation, query rewriting, and execution of optimized queries on the candidate groups. In the warm cache setting, no single approach won on all the queries.
- The reification approach to represent quads, *i.e.*, RF was slower than RIQ, Jena TDB, and Virtuoso for most cases. This was largely due to the large number of triples that were produced due to the reification process as well as the increase in the number of triples patterns in the transformed queries.

6. Conclusions

RDF quads can aptly model the facts in a knowledge graph, which is becoming an important resource for users of the World Wide Web. Using SPARQL, rich queries can be expressed on a knowledge graph. In this paper, we presented our approach called RIQ for fast processing of SPARQL queries on large datasets containing RDF quads. RIQ employs a *decrease-and-conquer* approach to efficiently process SPARQL queries. It groups similar RDF graphs efficiently using a new vector representation and popular hashing techniques, and constructs a filtering index using a combination of BFs and CBFs for compactness. (Each group of similar RDF graphs is indexed separately.) To process a SPARQL query, RIQ first searches the filtering index to identify candidate groups that may contain results for the query. It then methodically rewrites the query and executes optimized queries on the candidates using an existing SPARQL processor (*e.g.*, Jena TDB, Virtuoso) to obtain the final results.

We conducted a comprehensive performance evaluation of RIQ using real and synthetic datasets, each containing about 1.4 billion quads. Through our experiments, we observed that RIQ's *decrease-and-conquer* approach enabled efficient query processing of high selectivity SPARQL queries that matched a small fraction of the named graphs in a dataset. The filtering index of RIQ was effective in pruning away a large number of named graphs that did not contain true matches for the queries. As a result, RIQ significantly outperformed its competitors like Virtuoso and Jena TDB for such queries when I/O was the dominating factor. On the other hand, when a query had low selectivity and matched a large number of named graphs in a dataset, RIQ's decrease-and-conquer approach was less effective than Virtuoso.

Acknowledgments

We are grateful to the anonymous reviewers for their comments and suggestions. This work was supported by the National Science Foundation under Grant No. 1115871.

Appendix A. Queries

BTC-2012 queries

```
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:

```
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" .
    ?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:

```
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 .
  }
}
```

B3:

```
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 .
  }
}
```

B4:

```
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 .
  }
}
```

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

B5:

```
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 .
  }
}
```

B6:

```
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 .
  }
}
```

B7:

```
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 .
  }
}
```

B8:

```
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 }
    }
}
```

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

B9:

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

B10:

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


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

B11:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
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 . }
    }
}
```

LUBM queries

```
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:

```
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> .
```

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

```
    ?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 .
  }
}
```

L2:

```
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 .
  }
}
```

L3:

```
SELECT *
WHERE {
graph ?g {
?student1 ub:undergraduateDegreeFrom ?undergradUni .
?student1 ub:memberOf ?dept .
?student2 ub:undergraduateDegreeFrom ?undergradUni .
?student1 ub:advisor ?professor .
?publication ub:publicationAuthor ?student1 .
?publication ub:publicationAuthor ?student2 .
?publication ub:publicationAuthor ?professor.
?professor ub:name                    ‘‘AssociateProfessor5" .
?professor ub:telephone               ?tpnu .
?professor ub:emailAddress            ?emAddr .
?professor ub:undergraduateDegreeFrom ?bsdg .
?professor ub:teacherOf               ?course .
?professor ub:worksFor                ?dept .
?professor ub:researchInterest        ?researchInt .
?professor ub:mastersDegreeFrom       ?msdg .
?professor ub:doctoralDegreeFrom      ?phddg .
?student1 rdf:type ub:GraduateStudent .
?dept rdf:type ub:Department .
?dept ub:subOrganizationOf <http://www.University10.edu> .
?student2 rdf:type ub:GraduateStudent .
?undergradUni rdf:type ub:University .
?publication rdf:type ub:Publication .
?student2 ub:memberOf ?dept .
}
}
```

L4:

```
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 <http://www.University0.edu> .
  }
}
```

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

L5:

```
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> .
  }
}
```

L6:

```
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 .
  }
}
```

L7:

```
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 .
  }
}
```

L8:

```
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 .
  }
}
```

L9:

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

L10:

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

L11:

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

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

Appendix B. Visualization of large BGPs

Fig. B.10 shows the visual representation of the large BGPs in queries B1–B2. Fig. B.11 shows the visual representation of the large BGPs in queries L1–L3.

References

[1] Resource Descrip. Framework. http://www.w3.org/RDF.
[2] C. Bizer, T. Heath, T. Berners-Lee, Linked data—The story so far, Int. J. Semant. Web Inf. Syst. 5 (3) (2009) 1–22.
[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, Z. Ives, DBpedia: A nucleus for a web of open data, in: Proc. of ISWC'07, 2007, pp. 11–15.
[4] D. Vrandecic, M. Krötzsch, Wikidata: a free collaborative knowledgebase, Commun. ACM 57 (10) (2014) 78–85.
[5] Pfizer. https://semanticweb.com/tag/pfizer.
[6] Have semantic technologies crossed the chasm yet? https://semanticweb.com/have-semantic-technologies-crossed-the-chasm-yet_b16484.
[7] The Knowledge Graph. http://www.google.com/insidesearch/features/search/knowledge.html.
[8] Facebook announces its third pillar graph search that gives you answers, not links like Google. http://techcrunch.com/2013/01/15/facebook-announces-its-third-pillar-graph-search/.
[9] Bing satori. http://searchengineland.com/library/bing/bing-satori.
[10] SPARQL 1.1. http://www.w3.org/TR/sparql11-query/.
[11] Seman. Web Challenge. http://challenge.semanticweb.org/.
[12] Linking Open Gov. Data. http://logd.tw.rpi.edu/.
[13] J. Hoffart, F.M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, G. Weikum, YAGO2: Exploring and querying world knowledge in time, space, context, and many languages, in: Proc. of WWW'11, 2011, pp. 229–232.
[14] D. Abadi, A. Marcus, S. Madden, K. Hollenbach, SW-Store: A vertically partitioned DBMS for Semantic Web data management, VLDB J. 18 (2) (2009) 385–406.
[15] C. Weiss, P. Karras, A. Bernstein, Hexastore: Sextuple indexing for Semantic Web data management, Proc. VLDB Endow. 1 (1) (2008) 1008–1019.
[16] T. Neumann, G. Weikum, The RDF-3X engine for scalable management of RDF data, VLDB J. 19 (1) (2010) 91–113.
[17] M. Atre, V. Chaoji, M.J. Zaki, J.A. Hendler, Matrix "Bit" loaded: A scalable lightweight join query processor for RDF data, in: Proc. of the 19th WWW Conference, 2010, pp. 41–50.
[18] J. Huang, D.J. Abadi, K. Ren, Scalable SPARQL querying of large RDF graphs, Proc. VLDB Endow. 4 (11) (2011) 1123–1134.
[19] M.A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, B. Bhattacharjee, Building an efficient RDF store over a relational database, in: Proc. of 2013 SIGMOD Conference, 2013, pp. 121–132.
[20] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, L. Liu, TripleBit: A fast and compact system for large scale RDF data, Proc. VLDB Endow. 6 (7) (2013) 517–528.
[21] K. Zeng, J. Yang, H. Wang, B. Shao, Z. Wang, A distributed graph engine for Web Scale RDF data, Proc. VLDB Endow. 6 (4) (2013) 265–276.
[22] V. Slavov, A. Katib, P. Rao, S. Paturi, D. Barenkala, Fast processing of SPARQL queries on RDF quadruples, in: Proc. of WebDB'14, 2014, pp. 1–6.
[23] P. Indyk, R. Motwani, Approximate nearest neighbors: Towards removing the curse of dimensionality, in: Proc. of the 13th ACM STOC, 1998, pp. 604–613.
[24] Jena TDB. http://jena.apache.org/documentation/tdb/.
[25] M.O. Rabin, Fingerprinting by Random Polynomials, Technical Report TR 15-81, Harvard University, 1981.
[26] A. Broder, On the resemblance and containment of documents. in: Proc. of the Compress. and Complex. of Sequences, 1997, pp. 21–29.
[27] M. Bawa, T. Condie, P. Ganesan, LSH forest: Self-tuning indexes for similarity search, in: Proceedings of the 14th International Conference on World Wide Web, Chiba, Japan, 2005, pp. 651–660.
[28] Q. Lv, W. Josephson, Z. Wang, M. Charikar, K. Li, Multi-probe LSH: Efficient indexing for high-dimensional similarity search, in: Proc. of the 33st VLDB Conference, Vienna, Austria, 2007, pp. 950–961.
[29] T.H. Haveliwala, A. Gionis, D. Klein, P. Indyk, Evaluating strategies for similarity search on the Web, in: Proc. of the 11th WWW Conference, 2002, pp. 432–442.
[30] P. Haghani, S. Michel, K. Aberer, Distributed similarity search in high dimensions using locality sensitive hashing, in: Proc. of the 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, 2009, pp. 744–755.
[31] A. Gupta, D. Agrawal, A.E. Abbadi, Approximate range selection queries in peer-to-peer systems, in: Conference on Innovative Data Systems Research, CIDR, 2003.
[32] V. Slavov, P. Rao, A gossip-based approach for Internet-scale cardinality estimation of Xpath queries over distributed semistructured data, VLDB J. 23 (1) (2014) 51–76.
[33] A. Broder, M. Mitzenmacher, Network applications of Bloom filters: A survey, Internet Math. 1 (4) (2003) 485–509.
[34] B. McBride, Jena: A Semantic Web toolkit, IEEE Internet Comput. 6 (2002) 55–59.
[35] K. Wilkinson, C. Sayers, H.A. Kuno, D. Reynolds, Efficient RDF storage and retrieval in Jena2, in: Proc. of SWDB'03, 2003, pp. 131–150.
[36] J. Broekstra, A. Kampman, F. van Harmelen, Sesame: A generic architecture for storing and querying RDF and RDF Schema, in: Proc. of ISWC'02, pp. 54–68.
[37] Virtuoso. http://lod.openlinksw.com/.
[38] Garlik 4store. http://4store.org/.
[39] AllegroGraph RDFStore. http://www.franz.com/agraph/allegrograph3.3/.
[40] Mulgara. http://www.mulgara.org/.
[41] A. Harth, J. Umbrich, A. Hogan, S. Decker, YARS2: A federated repository for querying graph structured data from the web, in: Proc. of ISWC'07/ASWC'07, Busan, Korea, 2007, pp. 211–224.
[42] D. Wood, P. Gearon, T. Adams, Kowari: A platform for Semantic Web storage and analysis, in: XTech 2005 Conference.
[43] S. Harris, N. Gibbins, 3store: Efficient bulk RDF storage, in: Practical and Scalable Semantic Systems, 2003.
[44] BigData: Presentation at OSCON 2008. http://bigdata.sourceforge.net/pubs/bigdata-oscon-7-23-08.pdf.
[45] Semantic Technologies Center, Oracle. http://www.oracle.com/technology/tech/semantic_technologies/index.html.
[46] E.I. Chong, S. Das, G. Eadon, J. Srinivasan, An efficient SQL-based RDF querying scheme, in: Proc. of the 31st VLDB Conference, 2005, pp. 1216–1227.
[47] Neo4j RDF. http://neo4j.org/.
[48] L. Ma, Z. Su, Y. Pan, L. Zhang, T. Liu, RStar: an RDF storage and query system for enterprise resource management, in: Proc. of CIKM'04, Washington, DC, USA, 2004, pp. 484–491.
[49] J.J. Levandoski, M.F. Mokbel, RDF data-centric storage, in: Proc. ICWS '09, Washington, DC, 2009, pp. 911–918.
[50] V. Bönström, A. Hinze, H. Schweppe, Storing RDF as a graph, in: Proceedings of the First Conference on Latin American Web Congress, Washington, DC, 2003, p. 27.
[51] R. Angles, C. Gutierrez, Querying RDF data from a graph database perspective, in: Proceedings of the Second European Semantic Web Conference, 2005, pp. 346–360.
[52] Y.H. Kim, B.G. Kim, J. Lee, H.C. Lim, The path index for query processing on RDF and RDF Schema, in: Advanced Communication Technology, 2005, ICACT 2005. The 7th International Conference on, vol. 2, 2005, pp. 1237–1240.
[53] A. Matono, T. Amagasa, M. Yoshikawa, S. Uemura, A path-based relational RDF database, in: ADC'05: Proceedings of the 16th Australasian database conference, Darlinghurst, Australia, 2005, pp. 95–103.
[54] R. Binna, W. Gassler, E. Zangerle, D. Pacher, G. Specht, SpiderStore: Exploiting main memory for efficient RDF graph representation and fast querying, in: Workshop on Semantic Data Management, Singapore, 2010.
[55] M. Janik, K. Kochut, BRAHMS: A WorkBench RDF store and high performance memory system for semantic association discovery, in: Proc. of ISWC'05, 2005, pp. 431–445.
[56] K. Wilkinson, Jena property table implementation, in: SSWS 2006, Athens, GA, 2006, pp. 35–46.
[57] J. Leeka, S. Bedathur, RQ-RDF-3X: going beyond triplestores, in: Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31–April 4, 2014, pp. 263–268.
[58] M. Sintek, M. Kiesel, RDFBroker: A signature-based high-performance RDF store, in: Proc.of ESWC'06, 2006, pp. 363–377.
[59] O. Udrea, A. Pugliese, V.S. Subrahmanian, GRIN: a graph based RDF index, in: Proc. of the 22nd National Conf. on Artificial Intelligence, 2007, pp. 1465–1470.
[60] M. Bröcheler, A. Pugliese, V.S. Subrahmanian, DOGMA: A disk-oriented graph matching algorithm for RDF databases, in: Proc. of ISWC'09, 2009, pp. 97–113.
[61] L. Zou, J. Mo, L. Chen, M.T. Özsu, D. Zhao, gStore: Answering SPARQL queries via subgraph matching, Proc. VLDB Endow. 4 (2011) 482–493.
[62] F. Picalausa, Y. Luo, G.H.L. Fletcher, J. Hidders, S. Vansummeren, A structural approach to indexing triples, in: Proc. of ESWC'12, 2012, pp. 406–421.
[63] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, N. Koziris, H2RDF+: An efficient data management system for big RDF graphs, in: Proc. of the 2014 ACM SIGMOD Conference, Snowbird, Utah, USA, 2014, pp. 909–912.
[64] S. Gurajada, S. Seufert, I. Miliaraki, M. Theobald, TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing, in: Proc. of the 2014 ACM SIGMOD Conference, Snowbird, Utah, USA, 2014, pp. 289–300.
[65] M. Hammoud, D.A. Rabbou, R. Nouri, S.-M.-R. Beheshti, S. Sakr, DREAM: Distributed RDF engine with adaptive query planner and minimal communication, Proc. VLDB Endow. 8 (6) (2015) 654–665.
[66] L. Chen, A. Gupta, M.E. Karul, Stack-based algorithms for pattern matching on DAGs, in: Proc. of the 31st VLDB Conference, Trondheim, Norway, Sept., 2005.
[67] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: Optimal XML pattern matching, in: Proc. of the 2002 ACM-SIGMOD Conference, Wisconsin, Madison, WI, 2002.
[68] J. Cheng, J. X. Yu, B. Ding, P.S. Yu, H. Wang, Fast graph pattern matching, in: Proc. of the 24th IEEE Intl. Conference on Data Engineering, Cancun, Mexico, April, 2008, pp. 913–922.
[69] L. Zou, L. Chen, M.T. Özsu, DistanceJoin: Pattern match query in a large graph database, PVLDB 2 (1) (2009) 886–897.
[70] J.R. Ullmann, An algorithm for subgraph isomorphism, J. ACM 23 (1) (1976) 31–42.
[71] R. Giugno, D. Shasha, GraphGrep: A fast and universal method for querying graphs, in: International Conference on Pattern Recognition, 2002.
[72] X. Yan, P. Yu, J. Han, Graph indexing: A frequent structure based approach, in: Proc. of the 2004 ACM-SIGMOD Conference, Paris, France, 2004.
[73] J. Cheng, Y. Ke, W. Ng, A. Lu, FG-index: Towards verification-free query processing on graph databases, in: Proc. of the 2007 ACM-SIGMOD Conference, Beijing, China, 2007, pp. 857–872.
[74] S. Zhang, M. Hu, J. Yang, TreePi: A novel graph indexing method, in: Proc. of the 23th IEEE Intl. Conference on Data Engineering, Istanbul, 2007, pp. 966–975.
[75] P. Zhao, J.X. Yu, P.S. Yu, Graph indexing: tree + delta >= graph, in: Proc. of the 33st VLDB Conference, 2007, pp. 938–949.

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005

[76] H. Shang, Y. Zhang, X. Lin, J.X. Yu, Taming verification hardness: An efficient algorithm for testing subgraph isomorphism, in: Proc. of the 34st VLDB Conference, Auckland, New Zealand, 2008, pp. 364–375.
[77] H. He, A.K. Singh, Closure-tree: An index structure for graph queries, in: Proc. of the 22th IEEE Intl. Conference on Data Engineering, Atlanta, 2006, pp. 38–49.
[78] D.W. Williams, J. Huan, W. Wang, Graph database indexing using structured graph decomposition, in: Proc. of the 23th IEEE Intl. Conference on Data Engineering, Istanbul, 2007, pp. 976–985.
[79] L. Zou, L. Chen, J.X. Yu, Y. Lu, A novel spectral coding in a large graph database, in: Proc. of the 11th Intl. Conference on Extending Database Technology, 2008.
[80] D. Pal, P.R. Rao, A tool for fast indexing and querying of graphs, in: Proceedings of the 20th International Conference Companion on World Wide Web, Hyderabad, India, 2011, pp. 241–244.
[81] D. Pal, P. Rao, V. Slavov, A. Katib, Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. System Sci. (2016).
[82] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, ACM Trans. Database Syst. 34 (3) (2009) 16:1–16:45.
[83] D. Beckett, Raptor. http://librdf.org/raptor/.
[84] Dablooms. https://github.com/bitly/dablooms.
[85] Y. Guo, Z. Pan, J. Heflin, LUBM: A benchmark for OWL knowledge base systems, Web Semant.: Sci. Serv. Agents 3 (2005) 158–182.
[86] M. Morsey, J. Lehmann, S. Auer, A.-C. N. Ngomo, DBPedia SPARQL benchmark: Performance assessment with real queries on real data, in: Proc. of the 10th International Conference on The Semantic Web, Bonn, Germany, 2011, pp. 454–469.

Please cite this article in press as: A. Katib, et al., RIQ: Fast processing of SPARQL queries on RDF quadruples, Web Semantics: Science, Services and Agents on the World Wide Web (2016), http://dx.doi.org/10.1016/j.websem.2016.03.005