Fast Processing of SPARQL Queries on RDF Quadruples

V. Slavov A. Katib V. Nuchimaniyanda P. Rao S. Paturi
D. Barenkala

University of Missouri-Kansas City, Kansas City

Technical Report UMKC TR-DB-2014-01

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 a conventional SPARQL processor that supports quads to produce the final results. We conducted a comprehensive evaluation of RIQ using a synthetic and a real dataset, each containing about 1.4 billion quads. Our results show that RIQ can outperform its competitors, namely, RDF-3X, Jena TDB, and Virtuoso, on a variety of SPARQL queries.

1 Introduction

The Resource Description Framework (RDF) is a standard model for data representation and interchange on the Web [13]. RDF uses URIs to name entities and their relationships. It enables easy merging of different data sources. While RDF was introduced in the late 90's as the data model for the Semantic Web, only in recent years, it has gained popularity on the Web. For example, Linked Data [24] exemplifies the use of RDF on the Web to represent different knowledge bases (*e.g.*, DBpedia [21]). Another example is Wikidata [56], 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 [12]), 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 [8], Facebook [4], and Microsoft [3] 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 [16], 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

```
@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> .                                      G1
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> .

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> .                                  G2
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> .
```

Figure 1: Dataset D containing RDF quads

graph. Moreover, there are datasets and knowledge bases on the Web such as Billion Triples Challenges [14], Linking Open Government Data (LOGD) [9], and Yago [39] which contain over a billion quads. One can view these datasets as a collection of RDF graphs. Using SPARQL's `GRAPH` keyword [16], a query can be posed to match a specific graph pattern within any single RDF graph.

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 [18, 57, 49, 20, 40, 26, 61, 62], 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 [49], 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 [54]. 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 [41] 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 [7]).

• RIQ achieved high performance on a synthetic and real-world dataset, each containing about 1.4 billion quads, on a variety of SPARQL queries. It outperformed RDF-3X and Jena TDB for queries containing large, complex graph patterns and achieved comparable performance for queries with small graph patterns. It also outperformed Virtuoso [17], a commercial tool, for queries with multiple graph patterns in the cold cache setting.

A preliminary version of this work appeared in the 17th International Workshop on the Web and Databases

(WebDB) 2014 [54].

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

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

Figure 2: Query Q

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 [16]. 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 Figure 1, which contains two RDF graphs G_1 and G_2. Consider a query Q shown in Figure 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.*

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 [47, 59], Sesame [30], Virtuoso [17], Garlik 4store [5], AllegroGraph [1], Mulgara [10], YARS2 [37], Kowari [60], 3Store [36], Bigdata(R) [2], Oracle 11g RDF [15, 31], Neo4j RDF [11]). 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 [47, 59, 36, 30, 31, 45, 44]. Some have attempted a graph based approach of storing and querying RDF [25, 19]; a few have taken a path-based approach by storing subgraphs in relational tables [43, 46]. 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 [23, 42].

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 [58] and used a column-oriented DBMS to achieve an order of magnitude performance improvement over previous techniques [18]. Recently, Neumann *et al.* developed RDF-3X [49] 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.* [57] developed Hexastore that also builds exhaustive indexes. However, Hexastore suffers from large index sizes due to lack of compression. Atre *et al.* [20] 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 during query processing.

More recently, Bornea *et al.* [26] 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.* [61] 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 [53, 55, 27, 63, 51]. 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 [40, 62, 50, 34, 35]. Huang *et al.* [40], 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 [62], 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+ [50] was proposed and builds eight indexes using HBase. It uses Hadoop to perform sort-merge joins during query processing. TriAD [34] 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 [35] 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 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 C.

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

4 The Design of RIQ

In this section, we present the novel design of RIQ (RDF Indexing on Quadruples). Figure 4 shows the architecture of RIQ. In this work, we deal with queries that conform to a subset of the SPARQL grammar [16] as shown in Figure 3.

```
Query => ’SELECT’ Variables ’WHERE’ ’{’ ’GRAPH’ Variables
      ’{’ GroupGraphPattern ’}’ ’}’ ResultModifiers
GroupGraphPattern => ’{’ GroupGraphPatternSub ’}’
GroupGraphPatternSub => TriplesBlock? GroupGraphPatternSubList*
GroupGraphPatternSubList => GraphPatternNotTriples ’.’? TriplesBlock?
GraphPatternNotTriples => GroupOrUnionGraphPattern | OptionalGraphPattern | Filter
GroupOrUnionGraphPattern =>   GroupGraphPattern ( ’UNION’ GroupGraphPattern )*
OptionalGraphPattern => ’OPTIONAL’ GroupGraphPattern
Filter => ’FILTER’ Constraint
Constraint => ’(’ expression ’)’ | BuiltInCall
BuiltInCall => ExistsFunction | NotExistsFunction
ExistsFunction => ’EXISTS’ GroupGraphPattern
NotExistsFunction => ’NOT EXISTS’ GroupGraphPattern
```

Figure 3: Subset of SPARQL grammar

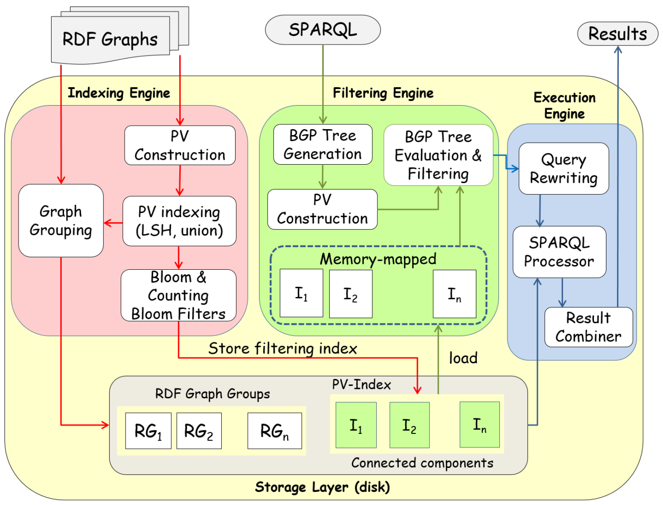


Figure 4: Overview of RIQ

4.1 Key Components of RIQ

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 a conventional 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.

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

Table 1: Transformations in RIQ

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. We denote the transformation on a triple (s,p,o) by $f_D : \mathbb{P} \times \{(s,p,o)\} \rightarrow O_D$, where the range O_D is shown 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. 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. Note that $\overline{V_c}$ requires space linear in the number of quads in the graph.

Algorithm 1 PV construction for an RDF graph

```
Input: An RDF graph $G$ with context c
Output: PV $\overline{V_c}$
1: for each $(s,p,o,c) \in G$ do
2:     for each $r \in \mathbb{P}$ do
3:         insert $\mathbb{H}(f_D(r,(s,p,o)))$ into $V_{c,r}$
4: for each $r \in \mathbb{P}$ do
5:     sort $V_{c,r}$
6: return $\overline{V_c}$
```

Our hash function $\mathbb{H}$ is based on Rabin's fingerprinting technique [52], which is efficient to compute. If we generate 32-bit hash values, the probability of collision is extremely low. Suppose the hash values are 32 bit unsigned integers and we use an irreducible polynomial of degree 31. If each triple pattern requires at most 2048 bits, then the probability of collision is less than 2^{-20} [28]. Thus, in practice, we can view $V_{c,SPO}$ as a set, because the quads/triples in a graph are always assumed to be unique. However, the remaining

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

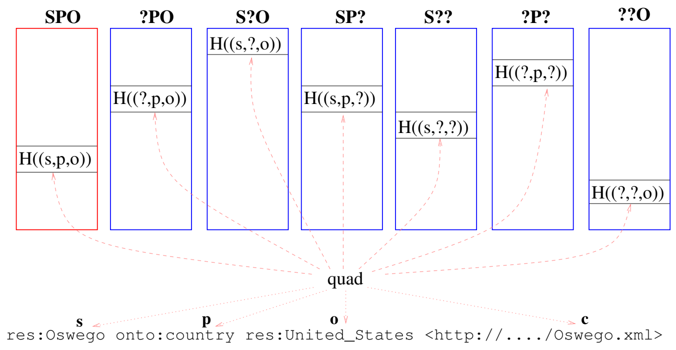
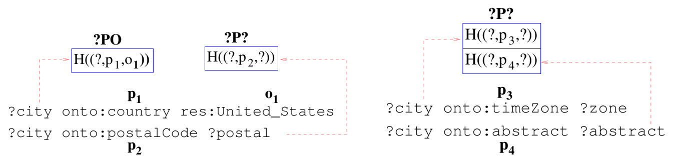


Figure 5: The PV of G_1. Note that s, p, and o, which appear inside H($\cdot$), should be replaced by their actual URIs.


?PO
H((?,p1,o1))
?P?
H((?,p2,?))
p1
o1
?city onto:country res:United_States
?city onto:postalCode ?postal
p2
?P?
H((?,p3,?))
H((?,p4,?))
p3
?city onto:timeZone ?zone
?city onto:abstract ?abstract
p4

(a) The PV of BGP_3 (b) The PVs of BGP_1

Figure 6: PVs of BGPs

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 Figure 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 triple patterns of the type `?s ?p ?o` during the PV construction.

Example 3 *Consider BGP_3 of query Q. As shown in Figure 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. Figure 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.

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

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

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 Figure 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 so that for a given BGP, we can quickly discard most of the non-matching RDF graphs. Second, we want to compactly store the PV-Index to minimize the cost of I/O during query processing. To address the first issue, we use the concept of locality sensitive hashing (LSH) [41]. For similarity on sets based on the Jaccard index, LSH on a set S, denoted by $\mathsf{LSH}_{k,l,m}(S)$ can be performed as follows [38]: Pick $k \times l$ random linear hash functions of the form $h(x) = (ax + b)\ mod\ u$, where u is a prime, and a and b are integers such that $0 < a < u$ and $0 \leq b < u$. Compute $g(S) = \min\{h(x)\}$ over all items in the set as the output hash value for S. Each group of l hash values is hashed (*e.g.*, using Rabin's fingerprinting) to the range $[0, m-1]$. This results in k hash values for S. It is known that given two sets S_1 and S_2 with similarity $p = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$, $\Pr[g(S_1) = g(S_2)] = p$. 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 is $1 - (1 - p^l)^k$. The above properties also hold for multisets.

Algorithm 3 The PV-Index Construction

```
Input: a list of PVs; $(k, l, m)$: LSH parameters; $\epsilon$: 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 graph
 2: for each PV $\overline{V}$ do
 3:     Add a new vertex $v_i$ to $\mathbb{V}$
 4:     for each $r \in \mathbb{P}$ do
 5:         $\{h_{i1}, ..., h_{ik}\} \leftarrow \mathsf{LSH}_{k,l,m}(V_r)$
 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 $\epsilon$ given the capacity $|U_{i,SPO}|$
13:     Construct a CBF for each of the remaining vectors of $U_i$ with false positive rate $\epsilon$ given the capacity
        $|U_{i,*}|$
14:     Store the ids of graphs belonging to $C_i$
15: return
```

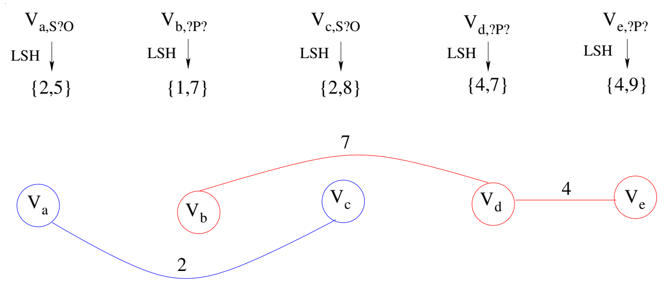


Figure 7: Grouping five PVs into two connected components shown in red and blue ($k = 2, m = 10$)

To address the second issue, we employ Bloom filters (BFs) and Counting Bloom filters (CBFs) [29] to compactly represent the PV-Index. A Bloom filter is a popular data structure to compactly represent a set of items and process membership queries on it. A Counting Bloom filter maintains n-bit counters instead of single bits and can represent multisets. Both BFs and CBFs can be configured to achieve a false positive rate based on their capacities [29].

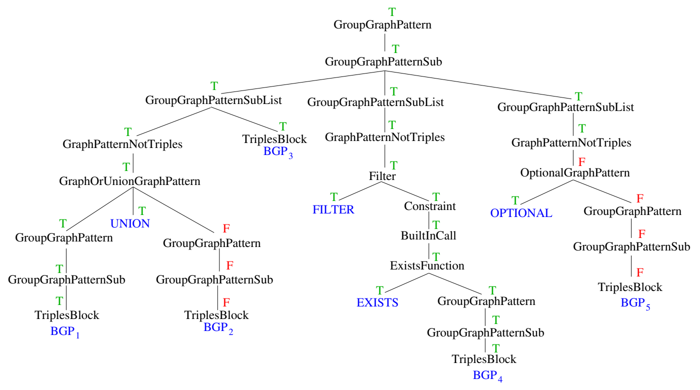
In Algorithm 3, we outline the steps to construct the PV-Index. We build a graph $\mathbb{G}$, where each vertex of $\mathbb{G}$ represents a PV. For every PV, we apply LSH on each of its seven vectors. Suppose there are two PVs such that the application of LSH on their vectors for the same pattern r, produces at least one identical hash value, then we add an edge between the vertices representing these PVs (Lines 2 to 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.)

Example 5 *Let us consider the example in Figure 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.*

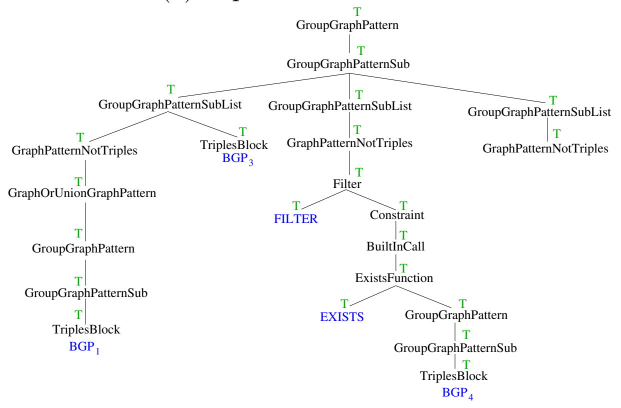
To compactly represent the union computed for a connected component, we use a combination of one Bloom filter (BF) and six Counting Bloom filters (CBFs). The vector for the canonical pattern SPO is stored using a BF and the others are stored using CBFs. Each filter of a vector is configured for a false 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 a conventional SPARQL query processor) to produce the final results.



(a) A processed BGP Tree



(b) Pruned BGP Tree

Figure 8: Query processing

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. For example, the query in Figure 2 is represented by the BGP Tree in Figure 8(a). We maintain a Boolean variable $eval[n]$ for each node n in the tree to denote the status of the evaluation on a connected component of the PV-Index. With $eval[n]$ = `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 child of `GroupGraphPatternSub` evaluates to `FALSE`, we skip processing the remaining children (Line 4), because the RDF graphs belonging to that connected component will not produce a match for the subexpression rooted at `GroupGraphPatternSub`. For `GroupOrUnionGraphPattern`, however, at least one of

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

```
1: Let $c_1, ..., c_\tau$ denote the child nodes of $n$ (left-to-right) ignoring those corresponding to braces
2: for $i = 1$ to $\tau$ do
3:   $eval[c_i] \leftarrow$ EvalBGPTree($c_i$, $j$)
4:   if $n$ is GroupGraphPatternSub & $eval[c_i]$ = FALSE then
5:     $eval[n] \leftarrow$ FALSE
6:     return FALSE {//skip rest of the nodes}
7: if $n$ is GroupOrUnionGraphPattern then
8:   $eval[n] \leftarrow \bigvee_{i=1}^{\tau} eval[c_i]$
9: else if $n$ is ExistsFunction then
10:   $eval[n] \leftarrow eval[c_\tau]$
11: else if $n$ is NotExistsFunction then
12:   $eval[n] \leftarrow$ TRUE
13: else if $n$ is Expression then
14:   $eval[n] \leftarrow$ TRUE {//skip processing predicates}
15: else if $n$ is TriplesBlock then
16:   Let $q$ denote the basic graph pattern
17:   $eval[n] \leftarrow$ IsMatch($q, j$)
18: else if $n$ is not a leaf then
19:   $eval[n] \leftarrow eval[c_\tau]$
20: else
21:   $eval[n] \leftarrow$ TRUE {//leaf nodes like UNION, FILTER}
22: if $n$ is OptionalGraphPattern then
23:   return TRUE
24: return $eval[n]$
```

its children *i.e.*, `GroupGraphPattern`, should evaluate to `TRUE` to produce a match (Line 7).

When a BGP is encountered (Line 15), 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. If `OptionalGraphPattern` evaluates to `FALSE`, we return `TRUE` because of the semantics of `OPTIONAL` in SPARQL. If $eval[root]$ = `TRUE`, then the group of RDF graphs belonging to that connected component is a candidate for further processing.

For the 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 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. The result modifiers and predicates within `FILTER` are included in the optimized query. All the projected variables in the original query are projected in the optimized query. Figure 8(b) shows the pruned BGP Tree for the example in Figure 8(a). Based on pruned BGP Tree, we generate the optimized query shown in Figure 9. In this query, the `OPTIONAL` block and one block in the `UNION` are absent. The optimized query can then be executed on the candidate using a tool like Jena TDB. The results from all the candidates are combined to produce the final output.

5 Performance Evaluation

In this section, we report the comprehensive performance evaluation of RIQ and compare it with RDF-3X, Jena TDB, and Virtuoso on real and synthetic datasets with about 1.4 billion RDF statements. We compared RIQ with the latest version of RDF-3X, Apache Jena 2.11.1 (TDB), and Virtuoso Open-Source Edition 7.1.0. RDF-3X and Virtuoso are written in C++. Jena TBD is a Java codebase. Also, Jena TDB and Virtuoso can index RDF quads and support queries with the `GRAPH` keyword. We ran all the experiments on a 64-bit Ubuntu 12.04 machine with 4 Intel Xeon 2.4GHz cores and 16GB RAM. RIQ, a C++ codebase, uses popular open-source libraries for parsing RDF data [22] and constructing BFs and CBFs [32].

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

```
1: For connected component $j$, let $\mathbb{F}_{j,r}$ denote the BF or CBF constructed for pattern $r$
2: for each $r \in \mathbb{P}$ do
3:   Construct $\mathbb{F}_{q,r}$ with the same capacity and false positive rate as $\mathbb{F}_{U_j,r}$
4: for each bit in $\mathbb{F}_{q,SPO}$ set to 1 do
5:   if the corresponding bit in $\mathbb{F}_{U_j,SPO}$ is 0 then
6:     return FALSE
7: for each $r \in \mathbb{P} \setminus \{SPO\}$ do
8:   for each non-zero counter in $\mathbb{F}_{q,r}$ do
9:     Let $c$ be the counter value
10:    if the corresponding counter in $\mathbb{F}_{U_j,r}$ is less than $c$ then
11:      return FALSE
12: return TRUE
```

Algorithm 6 PruneBGPTree(node n)

```
1: Let $c_1, ..., c_\tau$ denote the child nodes of $n$ (left-to-right) ignoring those corresponding to braces
2: if $eval[n]$ = FALSE then
3:   if $n$'s parent is NotExistsFunction then
4:     return TRUE
5:   else if $n$ is OptionalGraphPattern then
6:     return FALSE
7:   else if $n$ is GroupGraphPattern & left-sibling is UNION then
8:     Prune away the subtree rooted at the left-sibling of $n$
9:   Prune away the subtree rooted at $n$ from the BGP Tree
10: else
11:   for $i = 1$ to $\tau$ do
12:     $status \leftarrow$ PruneBGPTree($i$)
13:     if $status$ = FALSE then
14:       Prune away the subtree rooted at $i$ from the BGP Tree
15: return TRUE
```

5.1 Datasets and Queries

We used one synthetic and one real dataset in our experiments. The synthetic dataset was generated using the Lehigh University Benchmark (LUBM) [33] and contained 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 one RDF graph. The real dataset was BTC 2012 [14], 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 graphs.

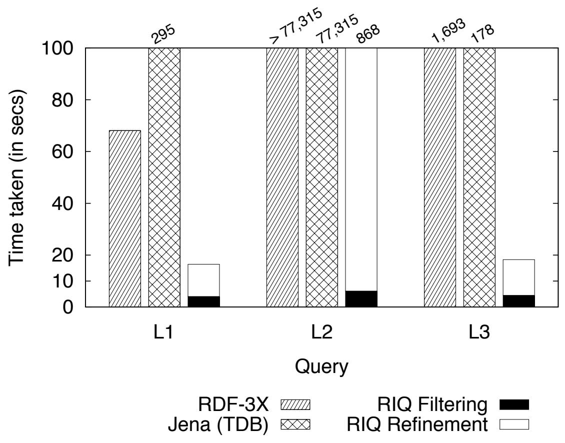
For LUBM, the query set included 3 SPARQL queries with large, complex BGPs (L1-L3) and 9 others (L4-L12) with small BGPs that are variations of the queries in the LUBM benchmark. For BTC 2012, the query set included 2 SPARQL queries with large, complex BGPs (B1, B2) and 5 others (B3-B7) with small BGPs. 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 [48]. The number of BGPs and triples patterns in each query and the number of results output for each query are shown in Table 2. (The queries are listed in B.)

5.2 Performance Evaluation on Queries with a Single BGP

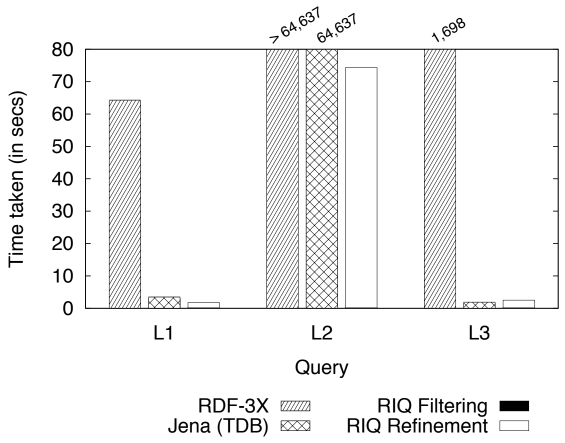
We conducted the first set of experiments to compare RIQ with its competitors for queries with a single BGP (*i.e.*, L1-L12 and B1-B7). Our goal was to demonstrate the effectiveness of RIQ's PV-Index and decrease-and-conquer strategy for efficient query processing. Because RDF-3X can only index RDF triples, we constructed a index on triples using Jena TDB for fair comparison. Unfortunately, in this scenario, Virtuoso failed to index the datasets on our machine with 16 GB of RAM. (It ran for a week and finally crashed.) So we

```
SELECT ?g ?city ?area ?code ?zone ?abstract ?postal
       ?offset ?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 . }
  }}
```

Figure 9: Optimized query



(a) Cold cache setting



(b) Warm cache setting

Figure 10: Time taken to process queries with large BGPs on LUBM

decided to drop Virtuoso from the comparison. RIQ constructed the PV-Index. Each group of RDF graphs was indexed by Jena TDB. The number of results output by each method matched the numbers reported in Table 2.

Table 2: Queries for LUBM and BTC 2012. 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.

Dataset	**Query**	**# of BGPs**	**# of triple patterns**	**# of results**
LUBM	L1 (l)	1	18	24
	L2 (l)	1	11	7,082
	L3 (l)	1	22	0
	L4 (s)	1	6	2,462
	L5 (s)	1	1	25,205,352
	L6 (s)	1	6	468,047
	L7 (s)	1	1	79,163,972
	L8 (s)	1	2	10,798,091
	L9 (s)	1	6	440,834
	L10 (s)	1	5	8,341
	L11 (s)	1	4	172
	L12 (s)	1	6	0
BTC 2012	B1 (l)	1	19	6
	B2 (l)	1	21	5
	B3 (s)	1	4	47,493
	B4 (s)	1	6	146,012
	B5 (s)	1	7	1,460,748
	B6 (s)	1	5	0
	B7 (s)	1	5	12,101,709
	B8 (m)	5	8	249,318
	B9 (m)	4	7	149,306
	B10 (m)	7	12	196
	B11 (m)	2	5	525,435

5.2.1 Index Construction

We report the index construction cost of RIQ on LUBM and BTC 2012. Note that the size of LUBM and BTC 2012 were 217 GB and 218 GB, respectively. 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. The PV-Index for LUBM and BTC 2012 had 487 and 526 connected components, respectively. Its size and the parameters used to tune the filters are also reported in the table. 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 graph groups during query processing.

RDF-3X indexed LUBM in 35,731 secs, and the index size was 77 GB. It indexed BTC 2012 in 35,995 secs, and the index size was 87 GB. Jena TDB indexed LUBM in 185,192 secs, and the index size was 121 GB. It indexed BTC 2012 in 119,805 secs, and the index size was 110 GB.

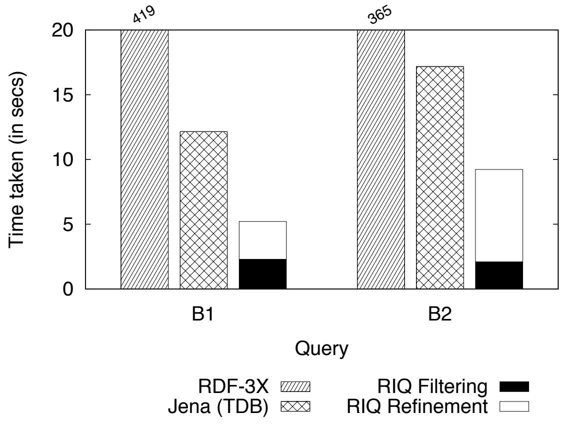
5.2.2 Query Processing

We measured the wall-clock time taken to process each query in both cold and warm cache settings, and report the average over 3 runs. We dropped the file system buffer cache by issuing the command `echo 3 > /proc/sys/vm/drop_caches`. (Jena TDB was executed with its default statistics-based optimization.)

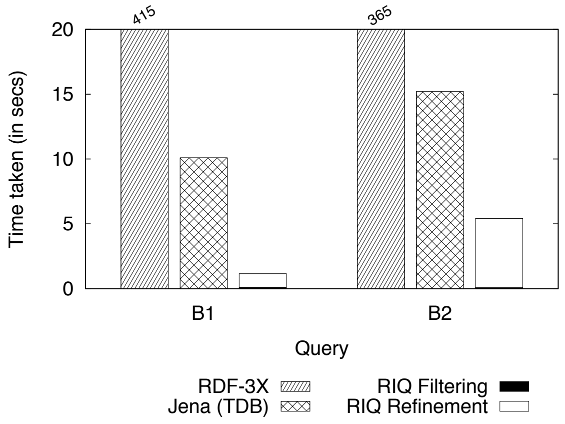
First, we show that the *decrease-and-conquer* approach of RIQ is more effective than the popular join-based processing (by first matching individual triple patterns) on queries with large, complex BGPs. All of the large, complex queries had at least one undirected cycle. For RIQ, we report both the filtering cost (using the PV-Index) and the refinement cost of processing the candidate groups to produce the final results. The results for LUBM are reported in Figure 10. Figure 10(a) shows the results for LUBM in the cold cache

Table 3: RIQ's index construction cost

Dataset	Construction time (in secs)				# of unions	False +ve rate (ϵ)	Max. filter capacity	PV-Index size
	PVs	Connected components	BF/CBFs	Total				
LUBM	15,249	22,711	3,402	41,362	487	1%	10 M	12 GB
BTC 2012	16,700	27,348	2,476	46,524	526	5%	1 M	6.5 GB



(a) Cold cache setting



(b) Warm cache setting

Figure 11: Time taken to process queries with large BGPs on BTC 2012

setting. RIQ processed queries with large, complex BGPs (L1-L3) significantly faster than RDF-3X and Jena TDB. For example, RIQ processed L2 in 868 secs but Jena TDB required 77,315 secs. RDF-3X required more than 77,315 secs to finish. Figure 10(b) shows the results for LUBM in the warm cache setting. Once again, RIQ processed the queries L1 and L2 faster than RDF-3X and Jena TDB. For example, RIQ processed L2 in 74 secs, and Jena TDB required 64,367 secs. RDF-3X was the slowest and required more than 64,367 secs to finish. For L3, Jena TDB was slightly faster than RIQ (1.9 secs vs 2.5 secs).

The results for BTC 2012 are reported in Figure 11. Figure 11(a) shows the results for BTC 2012 in the cold cache setting. RIQ was significantly faster than RDF-3X and Jena TDB in processing queries B1 and B2. For example, RIQ processed B2 in 7.2 secs, while Jena TDB and RDF-3X required 17.2 secs and 365 secs, respectively. Similar trend was observed in the warm cache setting as shown in Figure 11(b).

The filtering time of RIQ for each query on LUBM is reported in Table 4. The filtering time for each query on BTC 2012 is reported in Table 5. RIQ identified a maximum of 16 candidate groups for queries L1-L3 and 3 candidate groups for queries B1 and B2.

Table 4: RIQ's filtering times for LUBM (large BGP queries).

Query	**Cold cache** Time taken (in secs)	**Warm cache** Time taken (in secs)
L1	4.03	0.15
L2	6.13	0.15
L3	4.50	0.16

Table 5: RIQ's filtering time on BTC 2012 (large BGP queries).

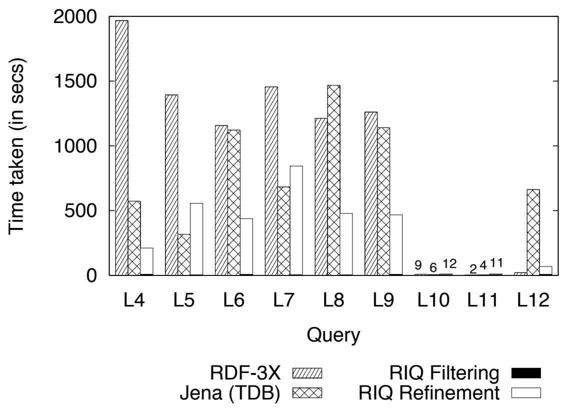
Query	**Cold cache** Time taken (in secs)	**Warm cache** Time taken (in secs)
B1	2.30	0.11
B2	2.10	0.08

Our next goal was to show that RIQ can achieve comparable performance with its competitors for queries with small BGPs. The tested queries (L4-L12 and B3-B7) contained less than 8 triple patterns. Figure 12(a) shows the results for LUBM in the cold cache setting. Interestingly, on LUBM, RIQ was faster than RDF-3X and Jena TDB for four out of the nine queries. RDF-3X was the fastest for two queries and Jena TDB for three queries. Figure 12(b) shows the results for LUBM in the warm cache setting. The trend was similar, and RIQ was fastest on four queries. Note that two queries where Jena TDB ran the fastest had very low selectivity (about 25 million and 79 million results). On BTC 2012, RIQ was significantly faster than its competitors in the cold cache setting for four out of the five queries. However, RDF-3X was the fastest in the warm cache setting for four out of the five queries.

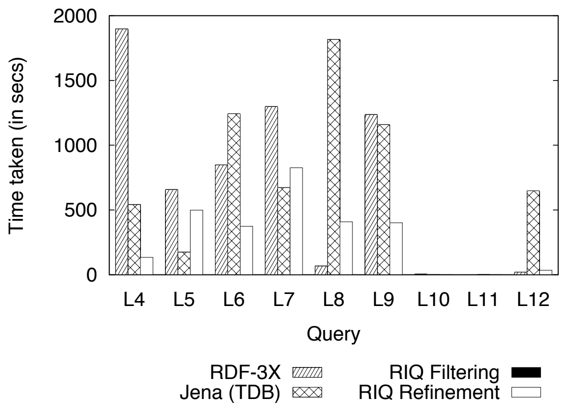
The filtering time of RIQ for L4-L12 is reported in Table 6. The filtering time of RIQ B3-B7 is reported in Table 7. One may notice that compared to a query with a large, complex BGP, a query with a small BGP had lower selectivity in many cases. In such a case, RIQ identified higher number of candidates groups during filtering for a BGP. As a result, the filtering time increased in many cases.

Table 6: RIQ's filtering times on LUBM (small BGP queries).

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



(a) Cold cache setting



(b) Warm cache setting

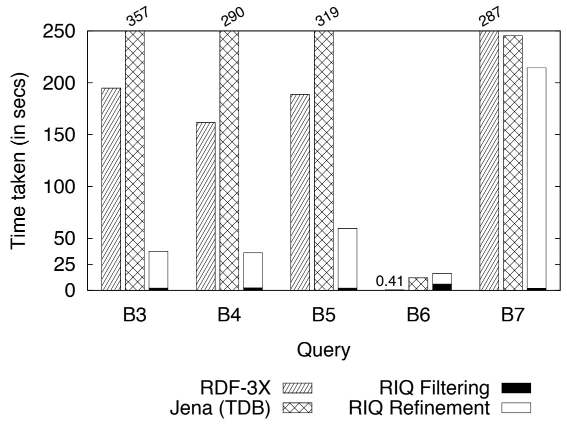
Figure 12: Time taken to process queries with small BGPs on LUBM

Finally, we compared the three approaches by computing the geometric mean of wall-clock time for the queries. Table 8 shows the geometric mean for queries on LUBM by considering L1-L3, L4-L12, and all the queries L1-L12. RIQ was the winner for all cases in both cold and warm cache settings. 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, RIQ was the winner in most cases except for the query set B3-B7.

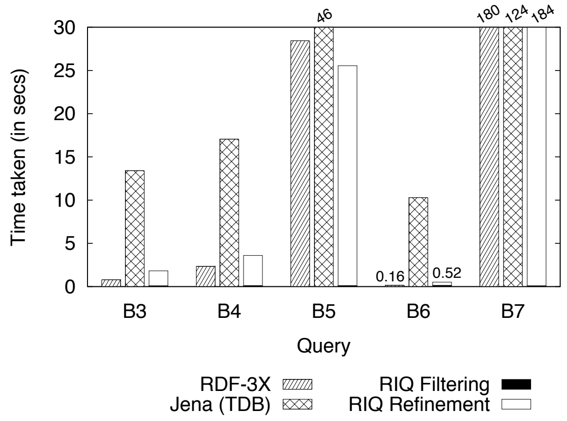
From the above results, we conclude RIQ's strategy of query processing yielded superior performance when the I/O cost was the dominating factor *i.e.*, in the cold cache setting. We report the winning approach for each query on LUBM and BTC 2012 in A. (See Tables 12 and 13).

5.3 Performance Evaluation on Queries with Multiple BGPs

We conducted the second 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. Jena TDB and Virtuoso were the competitors as they support queries with the `GRAPH` keyword. They were run using their default settings. RDF-3X does not support such queries and was therefore, dropped from the comparison.



(a) Cold cache setting



(b) Warm cache setting

Figure 13: Time taken to process queries with small BGPs on BTC 2012

5.3.1 Index Construction

Jena TDB constructed the index on BTC 2012 in 139,804 secs, and the index size was 275 GB. Virtuoso indexed the dataset in five and a half days, and the index size was 77 GB. RIQ's filtering index was the same as before. We only used BTC 2012, a real dataset, because of the lack of suitable queries with multiple BGPs for LUBM.

5.3.2 Query Processing

The nature of the tested queries are reported in Table 2. 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. Figure 14(a) shows the results for the cold cache setting. RIQ outperformed both Jena TDB and Virtuoso for all the four queries. On B10, RIQ showed the best improvement over Virtuoso and was about 2.4 times faster (16.3 secs vs 39.2 secs). On the other queries, RIQ was 1.3 to 1.5 times faster than Virtuoso. We consider this to be a remarkable achievement for RIQ, because Virtuoso is a commercial tool and has been heavily optimized over the years. Jena TDB was the slowest of the three approaches in the cold cache setting. For example, Jena TDB processed B9 in 648.9 secs, but RIQ executed the query in 110.7 secs. Figure 14(b) shows the results for the warm cache setting. RIQ was the fastest for B11 and processed the query in 76.7 secs. Jena TDB was the

Table 7: RIQ's filtering time on BTC 2012 (small BGP queries).

Query	**Cold cache** Time taken (in secs)	**Warm cache** Time taken (in secs)
B3	2.15	0.10
B4	2.28	0.11
B5	2.14	0.11
B6	6.05	0.14
B7	2.09	0.10

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

Query	**Cold cache** Geo. Mean (in secs)			**Warm cache** Geo. Mean (in secs)		
	RIQ	**RDF -3X**	**Jena TDB**	RIQ	**RDF -3X**	**Jena TDB**
L1-L3	**64.0**	2073.9	1595.8	**6.8**	1917.9	74.9
L4-L12	**163.7**	235.9	250.2	**65.1**	116.0	188.3
L1-L12	**129.5**	406.2	397.6	**37.1**	233.9	149.5

fastest for B8 and B9 and processed these queries in 38.7 secs and 33.3 secs, respectively. Virtuoso was the winner for B10 and processed the query in 0.16 secs. To summarize, RIQ's decrease-and-conquer approach yielded superior performance especially when I/O was the dominating factor during query processing, *i.e.*, cold cache setting.

The filtering times of RIQ for B8-B11 are shown in Table 10. Since each query had more than one BGP, the total filtering time was usually higher than for the other queries on BTC 2012. 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 secs for each query.

The geometric means are shown in Table 11. RIQ was the winner in the cold cache setting, and Virtuoso was the winner in the warm cache setting.

5.4 Summary of Results

Below we summarize the key findings of our performance evaluation.

- RIQ outperformed RDF-3X and Jena TDB on SPARQL queries with large, complex BGPs. The filtering index *i.e.*, the PV-Index provided significant advantage during query processing and enabled the decrease-and-conquer approach to process queries faster than previous approaches that rely on a large number of joins for finding the matches for a BGP.

- RIQ had comparable performance with RDF-3X and Jena TDB on SPARQL queries with small BGPs. RIQ had better performance than its competitors mainly in the cold cache setting.

- In the cold cache setting, RIQ outperformed Jena TDB and Virtuoso–a commercial tool–on queries with multiple BGPs combined using constructs like `UNION` and `OPTIONAL`. This was because of 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.

Table 9: Geometric mean of the query processing times for BTC 2012. The winning approach is shown in bold within shaded cells.

Query	**Cold cache** Geo. mean (in secs)			**Warm cache** Geo. mean (in secs)		
	RIQ	**RDF -3X**	**Jena TDB**	RIQ	**RDF -3X**	**Jena TDB**
B1-B2	**6.9**	391.3	14.6	**2.5**	389.1	12.3
B3-B7	**48.8**	58.7	157.6	6.9	**4.3**	26.7
B1-B7	**27.9**	100.9	79.9	**5.1**	15.6	21.4

Table 10: RIQ's filtering time on BTC 2012 (multiple BGP queries).

Query	**Cold cache** Time taken (in secs)	**Warm cache** Time taken (in secs)
B8	5.15	0.85
B9	5.12	0.78
B10	6.42	0.66
B11	6.21	0.61

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 are 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 a conventional SPARQL processor (*e.g.*, Jena TDB) 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 enables efficient SPARQL query processing on large RDF datasets. It significantly outperformed its competitors like RDF-3X and Jena TDB for queries with large, complex BGPs. This demonstrates that the decrease-and-conquer approach of RIQ leads to faster query processing than conventional approaches that rely on a large number of joins to find the matches for a BGP. RIQ also achieved comparable performance with RDF-3X and Jena TDB for queries with small BGPs. On queries with multiple BGPs, RIQ outperformed both Jena TDB and Virtuoso in the cold cache setting. Overall, RIQ yielded superior performance on a variety of SPARQL queries.

Acknowledgments

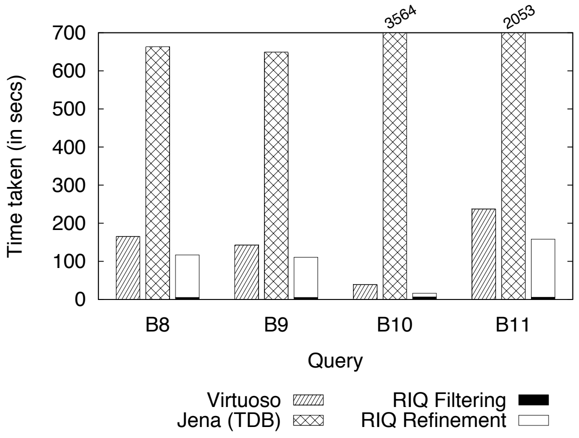
This work was supported by the National Science Foundation under Grant No. 1115871.

References

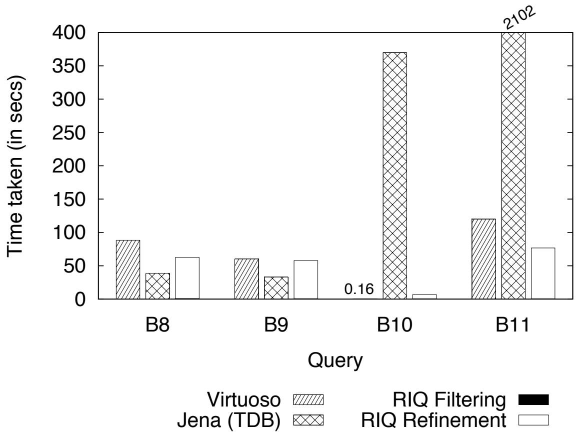
[1] AllegroGraph RDFStore. `http://www.franz.com/agraph/allegrograph3.3/`.

[2] BigData: Presentation at OSCON 2008. `http://bigdata.sourceforge.net/pubs/bigdata-oscon-7-23-08.pdf`.

[3] Bing satori. `http://searchengineland.com/library/bing/bing-satori`.



(a)



(b)

Figure 14: Queries with multiple BGPs

[4] 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/`.

[5] Garlik 4store. `http://4store.org/`.

[6] Have semantic technologies crossed the chasm yet? `https://semanticweb.com/have-semantic-technologies-crossed-the-chasm-yet_b16484`.

[7] Jena TDB. `http://jena.apache.org/documentation/tdb/`.

[8] The knowledge graph. `http://www.google.com/insidesearch/features/search/knowledge.html`.

[9] Linking Open Gov. Data. `http://logd.tw.rpi.edu/`.

[10] Mulgara. `http://www.mulgara.org/`.

[11] Neo4j RDF. `http://neo4j.org/`.

[12] Pfizer. `https://semanticweb.com/tag/pfizer`.

[13] Resource Descrip. Framework. `http://www.w3.org/RDF`.

[14] Seman. Web Challenge. `http://challenge.semanticweb.org/`.

Table 11: Geometric mean for BTC 2012. Best results are shown in bold within shaded cells.

Cold cache Time taken (in secs)			**Warm cache** Time taken (in secs)		
RIQ	**Jena TDB**	**Virtuoso**	RIQ	**Jena TDB**	**Virtuoso**
76.1	1,331.8	121.8	37.2	178.0	**17.9**

[15] Semantic Technologies Center, Oracle. `http://www.oracle.com/technology/tech/semantic_technologies/index.html`.

[16] SPARQL 1.1. `http://www.w3.org/TR/sparql11-query/`.

[17] Virtuoso. `http://lod.openlinksw.com/`.

[18] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: A vertically partitioned DBMS for semantic web data management. *VLDB Journal*, 18(2):385–406, 2009.

[19] R. Angles and C. Gutierrez. Querying RDF Data from a Graph Database Perspective. In *Proceedings of the Second European Semantic Web Conference*, pages 346–360, 2005.

[20] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" loaded: A scalable lightweight join query processor for RDF data. In *Proc. of the 19th WWW Conference*, pages 41–50, 2010.

[21] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, and Z. Ives. DBpedia: A nucleus for a web of open data. In *Proc. of ISWC '07*, pages 11–15, 2007.

[22] D. Beckett. Raptor. `http://librdf.org/raptor/`.

[23] R. Binna, W. Gassler, E. Zangerle, D. Pacher, and G. Specht. SpiderStore: Exploiting Main Memory for Efficient RDF Graph Representation and Fast Querying. In *Workshop on Semantic Data Management*, Singapore, 2010.

[24] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The story so far. *Int. Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[25] V. Bönström, A. Hinze, and H. Schweppe. Storing RDF as a Graph. In *Proceedings of the First Conference on Latin American Web Congress*, page 27, Washington, DC, 2003.

[26] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *Proc. of 2013 SIGMOD Conference*, pages 121–132, 2013.

[27] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. DOGMA: A disk-oriented graph matching algorithm for RDF databases. In *Proc. of ISWC '09*, pages 97–113, 2009.

[28] A. Broder. On the resemblance and containment of documents. In *Proc. of the Compress. and Complex. of Sequences*, pages 21–29, 1997.

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

[30] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proc. of ISWC '02*, pages 54–68.

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

[32] Dablooms. `https://github.com/bitly/dablooms`.

[33] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3:158–182, October 2005.

[34] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *Proc. of the 2014 ACM SIGMOD Conference*, pages 289–300, Snowbird, Utah, USA, 2014.

[35] M. Hammoud, D. A. Rabbou, R. Nouri, S.-M.-R. Beheshti, and S. Sakr. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *Proc. VLDB Endow.*, 8(6):654–665, Feb. 2015.

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

[37] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data From the Web. In *Proc. of ISWC'07/ASWC'07*, pages 211–224, Busan, Korea, 2007.

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

[39] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum. YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages. In *Proc. of WWW '11*, pages 229–232, 2011.

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

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

[42] M. Janik and K. Kochut. BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In *Proc. of ISWC '05*, pages 431–445, 2005.

[43] Y. H. Kim, B. G. Kim, J. Lee, and 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*, volume 2, pages 1237–1240, 2005.

[44] J. J. Levandoski and M. F. Mokbel. RDF Data-Centric Storage. In *Proc. ICWS '09*, pages 911–918, Washington, DC, 2009.

[45] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar: an RDF storage and query system for enterprise resource management. In *Proc. of CIKM '04*, pages 484–491, Washington, D.C., USA, 2004.

[46] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. A path-based relational RDF database. In *ADC '05: Proceedings of the 16th Australasian database conference*, pages 95–103, Darlinghurst, Australia, 2005.

[47] B. McBride. Jena: A Semantic Web Toolkit. *IEEE Internet Computing*, 6:55–59, 2002.

[48] M. Morsey, J. Lehmann, S. Auer, and 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*, pages 454–469, Bonn, Germany, 2011.

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

[50] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H2RDF+: An Efficient Data Management System for Big RDF Graphs. In *Proc. of the 2014 ACM SIGMOD Conference*, pages 909–912, Snowbird, Utah, USA, 2014.

[51] F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. A Structural Approach to Indexing Triples. In *Proc. of ESWC '12*, pages 406–421, 2012.

[52] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR 15-81, Harvard University, 1981.

[53] M. Sintek and M. Kiesel. RDFBroker: A signature-based high-performance RDF store. In *Proc.of ESWC '06*, pages 363–377, 2006.

[54] V. Slavov, A. Katib, P. Rao, S. Paturi, and D. Barenkala. Fast Processing of SPARQL Queries on RDF Quadruples. In *Proc. of WebDB '14*, pages 1–6, 2014.

[55] O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: a graph based RDF index. In *Proc. of the 22nd National Conf. on Artificial Intelligence*, pages 1465–1470, 2007.

[56] D. Vrandecic and M. Krtzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[57] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for Semantic Web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, 2008.

[58] K. Wilkinson. Jena property table implementation. In *SSWS 2006*, pages 35–46, Athens, GA, 2006.

[59] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proc. of SWDB'03*, pages 131–150, 2003.

[60] D. Wood, P. Gearon, and T. Adams. Kowari: A Platform for Semantic Web Storage and Analysis. In *XTech 2005 Conference.*

[61] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: A fast and compact system for large scale RDF data. *Proc. VLDB Endow.*, 6(7):517–528, 2013.

[62] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for Web Scale RDF data. *Proc. VLDB Endow.*, 6(4):265–276, Feb. 2013.

[63] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL queries via subgraph matching. *Proc. VLDB Endow.*, 4:482–493, May 2011.

A Summary of Winning Approaches

Table 12 shows the winning approach for queries on LUBM. Table 13 shows the winning approach for queries on BTC 2012.

Table 12: The winning approach is shown for queries on LUBM.

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

Table 13: The winning approach is shown for queries on BTC 2012.

Query	**Cold cache**	**Warm cache**
B1	RIQ	RIQ
B2	RIQ	RIQ
B3	RIQ	RDF-3X
B4	RIQ	RDF-3X
B5	RIQ	RIQ
B6	RDF-3X	RDF-3X
B7	RIQ	RDF-3X
B8	RIQ	Jena TDB
B9	RIQ	Jena TDB
B10	RIQ	Virtuoso
B11	RIQ	RIQ

B Queries

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 ?p ?c ?e ?ph ?res ?uguni ?msuni ?phduni ?s1n ?s2n ?s1 ?s2 ?pub WHERE { GRAPH ?g { ?s1 ub:advisor ? . ?s1 ub:name ?s1n . ?s1 rdf:type ub:UndergraduateStudent . ?s2 ub:advisor ?p . ?s2 ub:name ?s2n . ?s2 rdf:type ub:GraduateStudent . ?p rdf:type ub:FullProfessor . ?p ub:name "FullProfessor7" . ?p ub:teacherOf ?c . ?p ub:undergraduateDegreeFrom ?uguni . ?p ub:mastersDegreeFrom ?msuni . ?p ub:doctoralDegreeFrom ?phduni . ?p ub:worksFor `<http://www.Department17.`

University1001.edu> . ?p ub:emailAddress ?e . ?p ub:telephone ?ph . ?p ub:researchInterest ?res . ?pub ub:publicationAuthor ?p . ?pub ub:publicationAuthor ?s2 . } }

L2:

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

L3:

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

L4:

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

L5:

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

L6:

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

L7:

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

L8:

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

L9:

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

L10:

SELECT ?x ?y ?z WHERE { GRAPH ?g { ?x rdf:type ub:UndergraduateStudent . ?y rdf:type ub:Department . ?x ub:memberOf ?y . ?y ub:subOrganizationOf <http://www.University0.edu> . ?x ub:emailAddress ?z . } }

L11:

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

L12:

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

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"8sd:string . ?cnt2 geo:hasMember ?ctry2 . ?cnt2 rdf:type geo:geographical_region . ?cnt2 geo:nameShortEN "Asia"8sd: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 ?fperf ?actor ?film ?name ?rel WHERE { GRAPH ?g { ?fperf fbase:film.performance.actor ?actor . ?fperf fbase:film.performance.film ?film . ?film fbase:type.object.name ?name . ?film fbase:film.film.initial_release_date ?rel . } }

B4:

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

B5:

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

B6:

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

B7:

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

B8:

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 . } } }

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 rdfs: `<http://www.w3.org/2000/01/rdf-schema#>` PREFIX geo: `<http://www.w3.org/2003/01/geo/wgs84_pos#>` PREFIX foaf: `<http://xmlns.com/foaf/0.1/>`

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

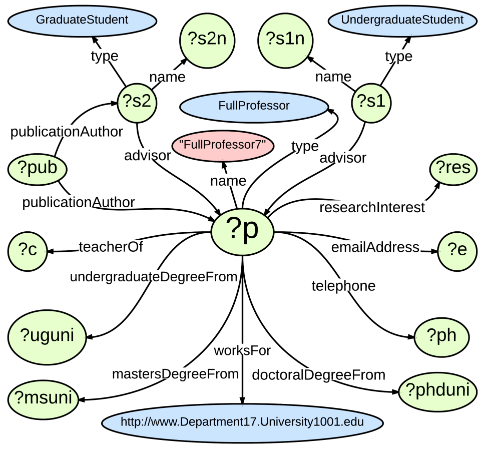
B11:

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#>`

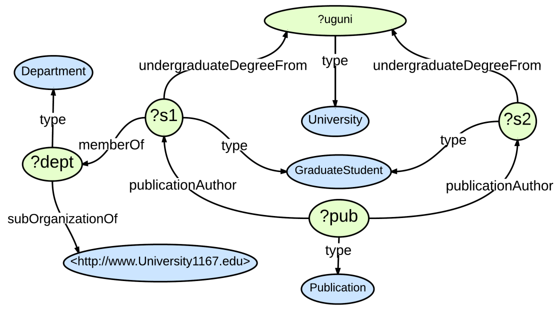
SELECT * WHERE { GRAPH ?g { ?var5 dbpedia-owl:thumbnail ?var4 . ?var5 rdf:type dbpedia-owl:Person . ?var5 rdfs:label ?var . ?var5 foaf:page ?var8 . OPTIONAL { ?var5 foaf:homepage ?var10 . } } }

C Visualization of Large BGPs

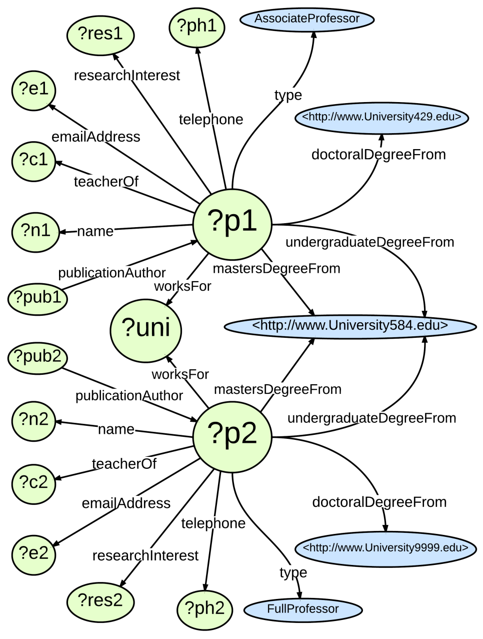
Figure 15 shows the visual representation of the large BGPs in queries L1-L3. Figure 16 shows the visual representation of the large BGPs in queries B1-B2.



(a)

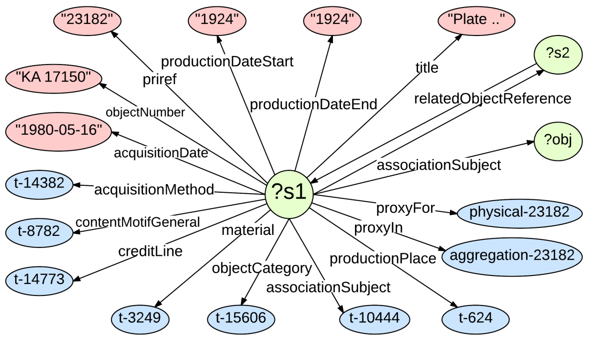


(b)

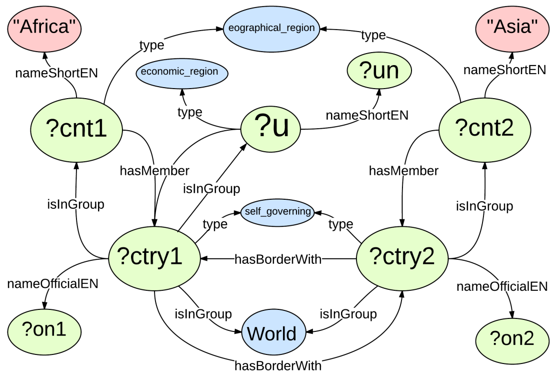


(c)

Figure 15: Visual representation of LUBM queries with large, complex BGPs



(a)



(b)

Figure 16: Visual representation of BTC queries with large, complex BGPs