UNIVERSITÀ DEGLI STUDI DI ROMA TRE
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

Answering SPARQL queries via Tensor Calculus

ROBERTO DE VIRGILIO AND FRANCO MILICCHIO

RT-DIA-187 **July 2011**

Dipartimento di Informatica e Automazione
Università di Roma Tre
{devirgilio,milicchio}@dia.uniroma3.it

ABSTRACT

We are witnessing the evolution of the Web from a worldwide information space of linked documents to a global knowledge base, composed of semantically interconnected resources (to date, 25 billion RDF triples, interlinked by around 395 million RDF links). RDF comes equipped with the SPARQL language for querying data in RDF format. Using so-called *triple patterns* as building blocks, SPARQL queries search for specified patterns in the RDF data. Although many aspects of the challenges faced in large-scale RDF data management have already been studied in the database research community, current approaches provide centralized hard-coded solutions, with high consumption of resources; moreover, these exhibit very limited flexibility dealing with queries, at various levels of granularity and complexity (*e.g.* so-called *non-conjunctive queries* that use SPARQL's UNION or OPTIONAL).

In this paper we propose a general model for answering SPARQL queries based on the first principles of linear algebra, in particular on *tensorial calculus*. Leveraging our abstract algebraic framework, our technique allows both quick decentralized processing, and centralized massive analysis. Experimental results show that our approach, utilizing recent linear algebra techniques—tailored to performance and accuracy as required in applied mathematics and physics fields—can process analysis efficiently, when compared to recent approaches. In particular, we are able to carry out the required computations even in high memory constrained environments, such as on mobile devices. Moreover, when dealing with massive amounts of data, we are capable of exploiting recent parallel and distributed technologies, subdividing our tensor objects into sub-blocks, and processing them independently.

1 Introduction

Ever since Tim Berners Lee presented the design principles for Linked Data[1], the public availability of Semantic Web data has grown rapidly. Today, many organizations and practitioners are all contributing to the "Web of Data", building RDF repositories either from scratch or by publishing in RDF data stored in traditional formats. The adoption of ontology languages such as RDFS, FLORA and OWL supports this trend by providing the means to semantically annotate Web data with meta-data, enabling ontological querying and reasoning.

The Resource Description Framework (RDF for short) provides a flexible method for representing information on the Semantic Web [19]. All data items in RDF are uniformly represented as triples of the form *(subject, predicate, object)*, sometimes also referred to as *(subject, property, value)* triples. Spurred by efforts like the Linking Open Data project, increasingly large volumes of data are being published in RDF.

RDF comes equipped with the SPARQL [24] language for querying data in RDF format. Using so-called *triple patterns* as building blocks, SPARQL queries search for specified patterns in the RDF data. For example, to retrieve all persons in the RDF data that have family name "Lee" together with their homepage we would write:

```
SELECT ?x ?y
WHERE ?x foaf:familyName "Lee" .
      ?x foaf:homepage ?y .
```

Many aspects of the challenges faced in large-scale RDF data management have already been studied in the database research community, including: native RDF storage layout and index structures [29, 1, 13, 20, 27]; SPARQL query processing and optimization [20, 21, 8]; as well as formal semantics and computational complexity of SPARQL [23, 25]. In particular, current research in SPARQL pattern processing (e.g., [20, 21, 28]) focuses on optimizing the class of so-called *conjunctive* patterns (possibly with filters) under the assumption that these patterns are more commonly used than the others.

Despite the fact that storing and querying large datasets of semantically annotated data in a flexible and efficient way represents a challenging area of research and a profitable opportunity for industry [11], semantic applications using RDF and linked-data repositories set performance and flexibility requirements that, in our opinion, have not yet been satisfied by the state of the art of data-management solutions. In particular, a *conjunctive pattern with filters* (CPF pattern for short) is a pattern that uses only the operators AND and FILTER. Although a sizable amount of SPARQL queries (*i.e.* 52%) consists of CPF patterns, non-CPF patterns (*i.e.* so-called *non-conjunctive* patterns with filters) that use SPARQL's UNION and OPTIONAL operators are more than substantial (*i.e.* 48%). While UNION and OPTIONAL correspond to the relational database operators OUTER UNION and LEFT OUTER JOIN, respectively, it has been noted by Perez et al. [23] that optimization techniques developed for the latter in relational DBMSs cannot be applied to SPARQL patterns involving the former. Unfortunately, Perez et al. [23] and Schmidt et al. [25] show that query evaluation quickly becomes hard for patterns including the operators UNION or OPTIONAL: given the importance of UNION and OPTIONAL in practical patterns, more research for processing non-conjunctive patterns is required.

In this paper we propose a novel approach based on first principles derived from the linear algebra field. Matrix operations are invaluable tools in several fields, from engineering and

[1] http://linkeddata.org/

design, graph theory, or networking. Standard matrix approaches focus on a standard two-dimension space, while we extend the applicability of such techniques with the more general definition of *tensors*, a generalization of linear forms, usually represented by matrices. We may therefore take advantage of the vast literature, both theoretic and applied, regarding tensor calculus. Computational algebra is employed in critical applications, such as engineering and physics, and several libraries have been developed with two major goals: *efficiency* and *accuracy*.

Contribution

Leveraging such background, this paper proposes a general model of RDF graph, mirrored with a formal tensor representation and endowed with specific operators, allowing both quick decentralized and centralized massive analysis on large volumes of data (*i.e.* billions of triples). We allow high flexibility dealing with queries, at various levels of granularity and complexity (*i.e.* CPF and no-CPF). Our model and operations, inherited by linear algebra and tensor calculus, is therefore theoretically sound, and its implementation may benefit of several numerical libraries developed in the past. Additionally, due to the properties of our tensorial model, we are able to attain two significative features: the possibility of conducting computations in memory-constrained environments such as on mobile devices, and exploiting modern parallel and distributed technologies, owing to the possibility for matrices, and therefore tensors, to be dissected into several chunks, and processed independently. Finally we provide TENSORRDF, a Java tool implementing our approach.

Outline

Our manuscript is organized as follows. In Section 2 we will briefly recall the available literature, while Section 3 will be devoted to the introduction of tensors and their associated operations, in addition to a brief recall of RDF and SPARQL. The general model of an RDF graph, accompanied by a formal tensorial representation is supplied in Section 4, subsequently put into practice in Section 5, where we provide the reader a method of analyzing RDF data within our framework. We benchmark our approach with several test beds, and supply the results in Section 6. Finally, Section 7 sketches conclusion and future work.

2 Related Work

Existing systems for the management of Semantic-Web data can be discussed according to two major issues: *storage* and *querying*.

Considering the storage, two main approaches can be identified: the first focuses on developing native storage systems to exploit ad-hoc optimisations, while the second makes use of traditional DBMSs (such as relational and object-oriented). Generally speaking, native storage systems (such as AllegroGraph[2], OWLIM[3] or RDF-3X [20]) are more efficient in terms of load and update time, whereas the adoption of mature data management systems has the advantage of relying on consolidate and effective optimisations. Indeed, a drawback of native approaches consists in the need for re-thinking query-optimization and transaction-processing techniques.

[2] http://www.franz.com/products/allegrograph/

[3] http://www.ontotext.com/owlim/

Early approaches to Semantic-Web data management (e.g., RDFSuite [3], Sesame[4], KAON[5], TAP[6], Jena[7], Virtuoso[8] and the semantic extensions implemented in Oracle Database 11g R2 [7]) focused on the triple *(subject, predicate, object)* or quads *(graph, subject, predicate, object)* data models, which can be implemented as one large relational table with three, or four, columns. Queries are formulated in RDF(S)-specific languages (e.g., SPARQL [24], SeRQL[9], RQL[10] or RDQL[11]), translated into SQL and sent to the RDBMS. However, the number of required self-joins makes this approach infeasible in practice and the optimisations introduced to overcome this problem (such as property-tables [30], GSPO/OGPS indexing [12]) have proven to be query-dependent [9] or to introduce significant computational overhead [5]. Abadi et al. [9] have proposed the *Vertical Partitioning* approach, where the triple table is split into n two-column tables to exploit fast merge-joins for reconstructing all the information associated with resources. However, Sidirourgos et. al [26], have noticed that its hard-coded nature makes it difficult to generalise and use in practice.

On the querying side, in the earlier systems the efficiency of query processing depends only on the logical and physical organization of the data and on the query language complexity. Current research in SPARQL pattern processing (e.g., [20, 21, 28]) focuses on optimizing the class of so-called *conjunctive* patterns (possibly with filters) under the assumption that these patterns are more commonly used than the others. In particular, a *conjunctive pattern with filters* is a pattern that uses only the operators AND and FILTER. Nevertheless, Möller et al [18] were the first to analyze a log of SPARQL queries, studying (1) the types of SPARQL queries posed (SELECT, ASK, CONSTRUCT, DESCRIBE) and (2) the forms of triple patterns posed in practice. In the same way, Arias et al. [4] performed a similar analysis. Both works have observed that *non-conjunctive* queries that use SPARQL's UNION or OPTIONAL operators appear non-negligibly often practically, providing detailed statistics on the types of joins appearing in practical queries.

As we will demonstrate in Section 6, all the above approaches are optimized for centralized massive analysis, in particular focusing on SELECT SPARQL queries, requiring significant amount of resources (*i.e.* memory and space consumption). The efficiency on such proposals is strictly depending on the logical and physical organization, and on the complexity of the SPARQL query. Differently from the other approaches, TENSORRDF provides a *general-purpose* storage policy for RDF graphs. Our approach exploits linear algebra and tensor calculus principles to define an abstract model, independent from any logical or physical organization, natively represented by matrix. Moreover, as we will demonstrate in Section 6, such representation allows the possibility of conducting computations in memory-constrained environments such as on mobile devices, and exploiting modern parallel and distributed technologies.

---

[4] http://www.openrdf.org/

[5] http://kaon.semanticweb.org/.

[6] http://tap.stanford.edu/.

[7] http://jena.sourceforge.net/

[8] http://virtuoso.openlinksw.com/

[9] http://www.openrdf.org/doc/sesame/users/ch06.html

[10] http://139.91.183.30:9090/RDF/RQL/.

[11] http://www.w3.org/Submission/RDQL/.

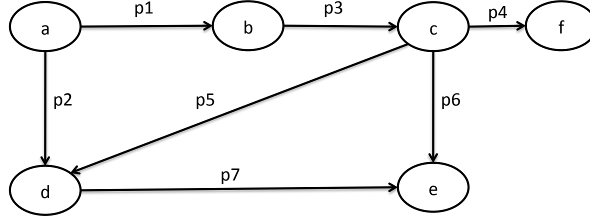


Figure 1: An example of RDF graph

3 Preliminary Issues

The following paragraphs will be devoted to a brief recall of RDF and SPARQL syntax, and tensor representations, which is preliminary to the introduction of our model in the ensuing sections.

RDF. Data in RDF is built from three disjoints sets $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$ of *IRIs*, *blank nodes*, and *literals*, respectively. For convenience we will use shortcuts like $\mathcal{IBL}$ and $\mathcal{IB}$ to denote the unions $\mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ and $\mathcal{I} \cup \mathcal{B}$, respectively.

All information in RDF is represented by triples of the form $\langle s, p, o \rangle$, where s is called the *subject*, p is called the *predicate*, and o is called the *object*. To be valid, it is required that $s \in \mathcal{IB}$; $p \in \mathcal{I}$; and $o \in \mathcal{IBL}$. Each set of RDF triples can easily be represented graphically as an edge-labeled graph in which the subjects and objects form the nodes, and edges are labeled by the corresponding predicates. For this reason, sets of RDF triples are also called *RDF graphs*. An *RDF dataset* D is a pair $D = (G, \gamma)$ with G an RDF graph and γ a function that assigns an RDF graph $\gamma(i)$ to each i in a finite set *dom*$(\gamma) \subseteq \mathcal{I}$ of IRIs. For instance Figure 1 shows an example of RDF graph. We do not use verbose IRIs for nodes and edges, but more simplified labels.

SPARQL. Abstractly speaking, a SPARQL query Q is a 5-tuple of the form $\langle qt, RC, DD, G_P, SM \rangle$, where qt is the *query type*, RC the *result clause*, DD the *dataset definition*, G_P the *graph pattern* and SM the *solution modifier*. At the heart of Q lies the *graph pattern* G_P that searches for specific subgraphs in the input RDF dataset. Its result is a (multi-)set of *mappings*, each of which associates variables to elements of $\mathcal{IBL}$. In particular the official SPARQL syntax considers operators `UNION`, `OPTIONAL`, `FILTER` and *concatenation* via a point symbol (.) to construct graph patterns. We consider G_P as a 4-tuple $\langle \mathcal{T}, f, OPT, U \rangle$. $\mathcal{T}$ is a set of *triple patterns*, that are just like triples, except that any of the parts of a triple can be replaced with a *variable*, *i.e.* a variable starts with a $?$ and can match any node (resource or literal) in the RDF dataset. Given two triples patterns t_1 and t_2, they present an *intersection* if they have a node in common (*i.e.* subject or object), and we will denote it as $t_1 \leftrightarrow t_2$. $\mathcal{T}$ is organized in disjunctive sets of triple patterns, *i.e.*, $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ where $\forall i, j : T_i \cap T_j = \emptyset$, with $i \neq j$. f is a `FILTER` constraint using boolean conditions to filter out unwanted query results. OPT is a set of `OPTIONAL` patterns trying to match a graph pattern, but does not fail the whole query if the optional match fails. U is a set of `UNION` statements in the form $(T_1, T_2, \ldots, T_m)$, where $T_i \in \mathcal{T}$. The *result clause* identifies which information to return from the query. It returns a table of variables (occurring in G_P) and values that satisfy the query. The *dataset definition*

is optional and specifies the input RDF dataset to use during pattern matching. If it is absent, the query processor itself determines the dataset to use. The optional *solution-modifier* allows sorting of the mappings obtained from the pattern matching, as well as returning only a specific window of mappings (e.g., mappings 1 to 10). The result is a list L of mappings. The actual output of the SPARQL query is then determined by the *query-type*:

- SELECT queries return projections of mappings from L (in order);
- ASK queries return a boolean: true if the graph pattern G_P could be matched in the input RDF dataset, and false otherwise;
- CONSTRUCT queries construct a new set of RDF triples based on the mappings in L; and
- DESCRIBE queries return a set of RDF triples that describes the IRIs and blank nodes found in L. The exact contents of this description is implementation-dependent.

We will not further describe the syntax and semantics of all constructs to use into a query, but refer instead to the SPARQL recommendation [24] for those components.

Example 1 We show three different SELECT queries.

```
Q1: SELECT ?x
    WHERE a ?y ?x. c ?z ?x.
          c ?p  f. b ?w  c.

Q2: SELECT ?y ?w
    WHERE a ?z ?x. ?y ?p ?x.
          FILTER (?y != a).
          OPTIONAL { ?x ?w e }

Q3: SELECT *
    WHERE {a ?z ?x}
          UNION
          {c ?p ?y}
```

Tensors. Tensors arise as a natural extension to linear forms. Linear forms on a vector space $\mathbb{V}$ defined over a field $\mathbb{F}$, and belonging to its *dual space* $\mathbb{V}^*$, are maps $\phi : \mathbb{V} \to \mathbb{F}$ that associate to each pair $\phi \in \mathbb{V}^*$, and $v \in \mathbb{V}$, an element $e \in \mathbb{F}$. This is usually denoted by the *pairing* $\langle \phi, v \rangle = e$, or by definition, with a *functional* notation, *i.e.*, $\phi(v) = e$. Such mapping exhibits the linearity property, *i.e.*, $\langle \phi, \alpha\, v + \beta w \,\rangle = \alpha \langle \phi, v \rangle + \beta \langle \phi, w \rangle$.

Generalizing the concept of linearity, we hence may introduce the following:

Definition 1 (Tensor) *A* tensor *is a multilinear form ϕ on a vector space $\mathbb{V}$,* i.e.*, a mapping*

$$\phi : \underbrace{\mathbb{V} \times \ldots \times \mathbb{V}}_{k-\text{times}} \longrightarrow \mathbb{F}\,, \tag{1}$$

the form ϕ assumes also the name of tensor of rank k *(or rank-k tensor).*

The property of *multilinearity* mirrors the previous definition of linearity: ϕ is, in fact, linear in each of its k domains.

As with linear forms, one possible representation of a tensor is with elements of the *matrix ring*. Therefore, a rank-k tensor may be represented by $M \in \mathbb{M}_{i_1 i_2 \ldots i_k}(\mathbb{F})$, that is a k-dimensional matrix with elements in $\mathbb{F}$.

A comprehensive description of tensors and their associated properties is beyond the scope of this manuscript; for a thorough review, see [2, 15], and [17].

Operations. Vectors and forms, both belonging to a *vector space*, provide different operations: from basic sum and product of a vector (form) with a scalar to more complex operations that may be defined on vector spaces. In particular, let us consider the following:

Definition 2 (Hadamard Product) *The* Hadamard product *of two vectors $v, w \in \mathbb{V}$ is the* entry-wise *product of their components,* i.e.*,*

$$v \circ w = r, \quad \textit{such that} \quad r_i := v_i w_i \,. \tag{2}$$

To clarify the above definition, we provide the reader with the following example:

Example 2 *Let us consider two vectors $v, w \in \mathbb{R}^3$, with $v = \begin{pmatrix} 1 & 4 & 0 \end{pmatrix}^t$, and $w = \begin{pmatrix} 2 & 3 & 9 \end{pmatrix}^t$, with $(\ )^t$ denoting the* transposed *vector. The Hadamard product will be*

$$v \circ w = \begin{pmatrix} 1 \cdot 2 & 4 \cdot 3 & 0 \cdot 9 \end{pmatrix}^t = \begin{pmatrix} 2 & 12 & 0 \end{pmatrix}^t \,.$$

Since a tensor of rank k will henceforth be denoted by a multidimensional matrix, we shall simplify notations by a variant of *Einstein's notation*. Such variant will be employed wherever a summation occurs: where we encounter two identical indices in a given expression, a summation on that index is implicitly defined.

Let us consider for example, a notable form, the *inner product* $(\cdot)$ between two vectors, known also as *scalar product*. We mention in passing that an inner product arises in defining a particular class of rank-2 tensors: symmetric (or Hermitian) positive-definite forms. If we consider a standard cartesian metrics, *i.e.*, an orthonormal basis [16], the inner product notation can be written leveraging the summation notation described above:

$$v \cdot w = \sum_j v_j w_j =: v_j w_j \,.$$

As with linear applications, represented by matrices, tensors may be applied to elements of vector spaces, giving rise the following:

Definition 3 (Application) *The* application *of a rank k tensor ϕ, represented by the matrix $M \in \mathbb{M}_{i_1 i_2 \ldots i_k}(\mathbb{F})$, to a vector $v \in \mathbb{V}$ is a rank $k-1$ tensor represented by the matrix $\widetilde{M}$:*

$$\widetilde{M}_{i_1 \ldots i_{j-1} i_{j+1} \ldots i_k} := M_{i_1 \ldots i_j \ldots i_k} v_{i_j} \,. \tag{3}$$

The reader should notice that the previous definition of *application* is a generalization of the common matrix-vector product. In other terms, applying a tensor to a vector simply "eats" one dimension, as evidenced by the indices in the previous definition.

Following a standard algebraic practice, indices may be rendered more intelligible when dealing with a tiny number of dimensions, *e.g.*, do not exceed 4 dimensions: in this case, instead of using $M_{i_1 i_2 i_3 i_4}$, we will employ the easier notation of M_{ijkl}.

Example 3 *Let us consider a rank-3 tensor ϕ represented by the matrix $M \in \mathbb{M}_{333}(\mathbb{R})$ and a vector $v \in \mathbb{R}^3$:*

$$M = \left(\begin{pmatrix} 1\\1\\1\\2\\2\\2\\3\\3\\3 \end{pmatrix} \begin{pmatrix} 1\\1\\1\\2\\2\\2\\3\\3\\3 \end{pmatrix} \begin{pmatrix} 1\\1\\1\\2\\2\\2\\3\\3\\3 \end{pmatrix} \right), \; v = \begin{pmatrix} 0\\1\\0 \end{pmatrix} .$$

The result of the application of ϕ to v is a rank-2 tensor

$$\langle \phi, v \rangle = M_{ijk} v_i = \begin{pmatrix} 2 & 2 & 2\\ 2 & 2 & 2\\ 2 & 2 & 2 \end{pmatrix} .$$

We mention in passing that applications of tensors span beyond simple vectors, with roots in Grassmann algebra, *i.e.*, the algebra of *multidimensional vectors* (cf. [17]).

Definition 4 (Kroneker Tensor) *The rank-k tensor $\delta_{i_1 i_2 \ldots i_k}$, whose matricial representation is $M_{i_1 i_2 \ldots i_k} = 1$ iff $i_1 = i_2 = \ldots = i_k$, is called* Kroneker tensor.

Kroneker tensor is well known in diverse fields ranging from computer science to economics, and it is commonly known as *Kroneker delta*. Moreover, with the canonical duality between forms and vectors [16], it is common to employ the very same symbol for a vector specification, *e.g.*, $v \in \mathbb{R}^3$, with $v = \delta_2 = (0 \; 1 \; 0)^t$. Note as the notation δ_2 does not contain any reference to the dimension, due to any lack of ambiguity, since here $\delta_2 \in \mathbb{R}^3$; this notation means that each component in a position different from 2 has value 0, while the component in position 2 has value 1.

The last definition we are going to introduce is a particular restriction of the general concept of maps between spaces, and a common notation in functional programming.

Definition 5 (Map) *The* map *is a function with domain a pair constituted by a function and a vector space, and codomain a second vector space:*

$$\text{map} : \mathcal{F} \times \mathbb{V} \rightarrow \mathbb{W} , \tag{4}$$

which associates to each component of a vector, the result of the application of a given function to the component itself:

$$\text{map}(f, v) = w, \quad w_i := f(v_i), \quad f \in \mathcal{F} ,$$

with $f : \mathbb{V} \rightarrow \mathbb{W}$, $v \in \mathbb{V}$, and $w \in \mathbb{W}$. Clearly, the space $\mathcal{F}$ is the space of functions with domain $\mathbb{V}$ and codomain $\mathbb{W}$, i.e.*, $\mathcal{F} = \{\kappa : \mathbb{V} \rightarrow \mathbb{W}\}$, as the reader noted.*

The *map* definition will be exemplified by the following:

Example 4 *Let us consider a vectors $v \in \mathbb{R}^3$, with $v = (0 \; 1/2\,\pi \; \pi)^t$. The* map *of the* sin *function on v results in*

$$\text{map}(\sin, v) = (\sin(0) \; \sin(1/2\,\pi) \; \sin(\pi))^t = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^t .$$

4 RDF Data Modeling

This section is devoted to the definition of a general model capable of representing all aspects of a given RDF graph. Our overall objective is to give a rigorous definition of a RDF graph and show how such representation is mapped within a standard tensorial framework.

4.1 A General Model

Let us define a set $\mathcal{E}$, with $\mathcal{E}$ being finite. A *property* of an element $e \in \mathcal{E}$ is defined as an application $\pi : \mathcal{E} \rightarrow \Pi$, where Π represents a suitable property codomain. Therefore, we define the application of a property $\pi(e) := \langle \pi, e \rangle$, *i.e.*, a property related to an element $e \in \mathcal{E}$ is defined by means of the pairing element-property; a property is a surjective mapping between e and its corresponding property value.

Formally, let us introduce the family of properties π_i, $i = 1, \ldots, n < \infty$, and their corresponding sets Π_i; we may therefore model the product set

$$\mathcal{E} \times \Pi_1 \times \ldots \times \Pi_n \, . \tag{5}$$

4.2 Tensorial Representation of a RDF graph

Given a RDF graph G, let us define the set $\mathcal{S}$ as the set of all resources (nodes) in G, with $\mathcal{S}$ being finite. A *property* of a resource $s \in \mathcal{S}$ is defined as an application $\pi : \mathcal{S} \rightarrow \Pi$, where Π represents a suitable property codomain. Therefore, we define the application of a property $\pi(s) := \langle \pi, s \rangle$, *i.e.*, a property related to a resource s is defined by means of the pairing resource-property; a property is a surjective mapping between a resource and its corresponding property value. Following the definition in the equation 5, a *RDF graph* ($\mathcal{G}$) is defined as the product set of all resources, and all the associated properties. Formally, let us introduce the family of properties π_i, $i = 1, \ldots, k + d < \infty$, and their corresponding sets Π_i; we may therefore model a RDF graph as the product set

$$\mathcal{G} = \underbrace{\mathcal{S} \times \Pi_1 \times \ldots \times \Pi_{k-1}}_{\mathbb{B}} \times \underbrace{\Pi_k \times \ldots \times \Pi_{k+d}}_{\mathbb{H}} \, . \tag{6}$$

We highlight the indices employed in the definition of $\mathcal{G}$. The reader should notice as we divided explicitly the first k spaces $\mathcal{S}, \Pi_1, \ldots, \Pi_{k-1}$, from the remaining ones. We have two different categories: *body* ($\mathbb{B}$) and *head* ($\mathbb{H}$). It should be hence straightforward to recognize the tensorial representation of $\mathcal{G}$:

Definition 6 (Tensorial Representation) *The* tensorial representation *of a RDF graph $\mathcal{G}$, as introduced in equation* (6)*, is a multilinear form*

$$\mathcal{G} : \mathbb{B} \longrightarrow \mathbb{H} \, . \tag{7}$$

A RDF graph can be therefore rigorously denoted as a rank-k tensor with values in $\mathbb{H}$. Now we can model some codomains Π with respect to the conceptual modeling provided above. In particular we define the property Π_1 as the set $\mathcal{O}$ of all resources in G (*i.e.* $\mathcal{S} \equiv \mathcal{O}$), Π_2 as the set of RDF properties $\mathcal{P}$ connecting a resource $s \in \mathcal{S}$ (*i.e.* the subject) with a resource $o \in \mathcal{O}$ (*i.e.* the object).

In the next paragraph we will illustrate how we organize such properties in $\mathbb{B}$ and $\mathbb{H}$. As a matter of fact, properties may *countable* or *uncountable*. By definition, a set A is *countable* if there exists a function $f : A \to \mathbb{N}$, with f being injective. For instance the sets $\mathcal{S}$, $\mathcal{O}$, $\mathcal{P}$ are countable. Therefore countable spaces can be represented with natural numbers and we can introduce a family of injective functions called *indexes*, defined as:

$$\mathrm{idx}_i : \Pi_i \longrightarrow \mathbb{N}\,, \qquad i \in [1, k+d]\,. \tag{8}$$

When considering the set $\mathcal{S}$, we additionally define a supplemental index, the *subject index function* $\mathrm{idx}_0 : \mathcal{S} \to \mathbb{N}$, consequently completing the map of all countable sets of $\mathcal{G}$ to natural numbers.

Implementation. Preliminary to describing an implementation of our model, based on tensorial algebra, we pose our attention on the practical nature of a RDF graph. Our treatment is general, representing a RDF graph with a tensor, *i.e.*, with a multidimensional matrix, due to the well known mapping between graphs and matrices (cf. [6]). However, a matrix-based representation need not to be *complete*. RDF graphs rarely exhibit completion: it is difficult to have an edge for each pairs of nodes (*i.e.* only if the graph is strongly connected). Hence, our matrix effectively requires to store only the information regarding connected nodes in the graph: as a consequence, we are considering sparse matrices [10], *i.e.*, matrices storing only non-zero elements. A sparse matrix may be indicated with different notations (cf. [10]), however, for simplicity's sake, we adopt the *tuple notation*. This particular notation declares the components of a tensor $M_{i_1 i_2 \ldots i_k}(\mathbb{F})$ in the form

$$\mathcal{M} = \Big\{ \{i_1 i_2 \ldots i_k\} \to f \neq 0\,, f \in \mathbb{F} \Big\}\,, \tag{9}$$

where we implicitly intended $f \neq 0 \equiv 0_{\mathbb{F}}$. As a clarifying example, consider a Kroneker vector $\delta_4 \in \mathbb{R}^5$: its sparse representation will be therefore constituted by a single tuple of one component with value 1, *i.e.*, $\{\ \{4\} \to 1\ \}$. In our case we have the notation $\mathbb{B} \longrightarrow \mathbb{H} = \mathbb{B}(\mathbb{H})$ where we consider $\mathbb{B} = \mathcal{S} \times \mathcal{O}$ and $\mathbb{H} = \mathcal{P}$.

Example 5 *Let us consider the example pictured in Figure 1, whose representative matrix is as follows:*

$$\begin{pmatrix} \cdot & p1 & \cdot & p2 & \cdot & \cdot \\ \cdot & \cdot & p3 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & p5 & p6 & p4 \\ \cdot & \cdot & \cdot & \cdot & p7 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

where, for typographical simplicity, we omitted $0_{\mathbb{H}} = (0)$, denoted with a dot. For clarity's sake, we outline the fact that $\mathcal{G}$ is a rank-2 tensor with dimensions 6×6 (i.e. the rows and columns corresponding to the nodes). In fact, we have six nodes in $\mathcal{S}$ and $\mathcal{O}$. Then we map the elements of $\mathcal{S}$ and $\mathcal{O}$ by using the index functions idx_0, and idx_1 (*i.e.* $\mathrm{idx}_0(a)$ = $\mathrm{idx}_1(a)$ = 1, $\mathrm{idx}_0(b)$ = $\mathrm{idx}_1(b)$ = 2, …, $\mathrm{idx}_0(f)$ = $\mathrm{idx}_1(f)$ = 6). In the sparse representation we have $\{\{1,2\} \to (p1), \{1,4\} \to (p2), \ldots, \{4,5\} \to (p7)\}$. In this way we mean that if $\{i,j\} \to (0)$ then there does not exist a triple between the subject with index i and the object with index j otherwise $\{i,j\} \to (p)$ means that the subject with index i is connected to the object with index

j by the RDF property p. In other terms, this implementation represents the adjacency matrix of the RDF graph in Figure 1, where rows are subjects and columns are objects in the triples (*i.e.* our model allows to express more information about triples). In the real implementation we use also the index function idx_2 for the properties; however for the sake of readability we will maintain the labels for properties in the rest of the paper.

4.3 Remarks

The reader noticed that all properties regarding spaces and their correlated operations, as defined in Sections 3 and 4, exhibit fundamental algebraic properties: *associativity*, *commutativity*, and *distributivity*. These characteristics help us in clarifying one implementative aspect regarding our sparse approach: memory constraints. At first sight, in order to perform operations such as tensor application, we may need to load the whole matrix and afterwards execute the required computation. Due to the distributivity of products with respect to summations, we actually may *split* a matrix in sub-matrices, carry out the application and finally, thanks to the associativity, add the partial results obtaining the correct solution. For example, if $A \in \mathbb{M}_{22}(\mathbb{R})$, and $v \in \mathbb{R}^2$ being a suitable vector, the matrix can be divided in two (or more) sub-matrices, thus giving the ability to perform computations independently:

$$Av = (A' + A'')v = A'v + A''v\,. \tag{10}$$

The above equation shows how the *sparsity* of a matrix—*i.e.*, the ratio between the overall dimensions and the actual non-zero stored elements—alleviates problems caused by the finiteness of computer memories. When performing $A'v$, we actually do not need any information regarding the subsequent A'' matrix and vice versa, and as a result, we may not store its elements in memory. Additionally, multiplication of a sparse matrix with a vector has a computational complexity that depends on the sparsity of the input: generally, very sparse matrices as the ones resulting from supply chains, yield *sub-quadratic* (cf. [10]), and even *quasi-linear* asymptotic complexity [14].

On a final note, due to the above mentioned properties, we are able to carry out the required computations with high memory constraints, such as on mobile devices. Moreover, when dealing with massive amounts of data, we are capable of exploit recent parallel and distributed technologies, remembering that tensor applications may be subdivided into sub-blocks, and processed independently.

5 SPARQL Answering

This section is devoted to the exemplification of our conceptual method, with the applicative objective of analyzing RDF data. With reference to the example provided in Section 4 (*i.e.* Figure 1 and *Example* 5), in the following we simplify our notation, employing a matrix M_{ij}: i referring to the index of subjects, and j being the index of objects. Such choice of attributes introduce no limitations on the generality of our approach, due to the fact that, in essence, applying a tensor to a vector means summing all the components sharing an index, regardless of ranks and dimensions.

5.1 SPARQL query features

Similarly to [18, 4], we analyzed a log of SPARQL queries harvested from the DBPedia SPARQL Endpoint from April to July 2010. The log contains 1343922 queries in total (after removal of syntactically invalid queries) posed by both humans and software robots. The log analysis produced interesting statistics on the usage of the different components in a query.

The four different types of SPARQL queries (SELECT, ASK, CONSTRUCT, DESCRIBE) are distributed as follows in the log:

Type	# in Log
SELECT	1295306 (96.35%)
ASK	39563 (2.94%)
CONSTRUCT	8179 (0.61%)
DESCRIBE	874 (0.10%)

Perhaps unsurprisingly, most queries in the log are SELECT queries, although a sizable amount of boolean ASK queries also occur. The amount of CONSTRUCT and DESCRIBE queries is negligible. A possible explanation for the fact that so few DESCRIBE queries occur may be that the semantics of DESCRIBE depends on the SPARQL processor, which makes it difficult for a user to know when such a query is useful.

Of the SELECT queries, 1232225 (96.35%) output all variables mentioned in their graph pattern. This indicates that practical queries do not introduce "temporary" variables that are not output but only used to constrain the possible values of the real output variables. Therefore FILTER constraints often refer to the variables in the result clause.

Only 309951 (23.06%) queries use a *dataset-definition* with a FROM (309925) or FROM NAMED (37) specifier. The rest of the queries does not specify a *dataset-definition* and hence evaluates their graph pattern with respect to the processor-defined RDF dataset.

SELECT, CONSTRUCT and ASK queries can use the ORDER BY, LIMIT, and OFFSET solution modifiers to sort the pattern matching results, or select only a specific window of these results. These modifiers are not used very often, as the following table shows:

Modifier	# in Log
ORDER BY	10238 (0.76%)
LIMIT	47248 (3.52%)
OFFSET	8711 (0.65%)

Finally, let us next analyze the structure of the graph patterns in the log, as follows

Operator o	# in Log
AND	665708 (49.53%)
FILTER	535450 (39.84%)
OPTIONAL	623370 (46.38%)
UNION	361251 (26.88%)
GRAPH	27 (0.00%)

In the table, we show for each pattern operator $o \in \{\text{AND}, \text{FILTER}, \text{OPTIONAL}, \text{UNION}, \text{GRAPH}\}$ the number of patterns in which o occurs. As such, it gives an idea of how frequent an operator is used in practical patterns. We see in particular that all operators except GRAPH occur non-negligibly often.

Such statistics allow us to simplify the features of a SPARQL query: in the following we will consider a SPARQL query Q as a 2-tuple of the form $\langle RC, G_P \rangle$, *i.e.* only SELECT queries with *result clause* and *graph pattern*, *i.e.* employing the operators {AND, FILTER, OPTIONAL, UNION}. This simplification does not compromise the feasibility and generality of the approach.

5.2 SPARQL tensor applications

A triple-pattern searches for specific subgraphs in the input RDF graph. We can comfortably perform the search efficiently, using the model described in Section 4, by applying the tensor application.

Therefore, given a triple-pattern $t_1 = \langle s_1, p_1, o_1 \rangle$, we can process t_1 by subject or by object. In the former case we search all subjects s connected to the object o_1, in the latter case we search all objects o connected to the subject s_1. If we process t_1 by the subject, given $j = \mathrm{idx}_1(o_1)$, we build a Kroneker vector as a vector δ_j, with $|\delta_j| = |\mathcal{O}|$, and finally apply of the rank-2 tensor represented by M_{ij} to δ_j, *i.e.*:

$$r = M_{ij}\delta_j \,.$$

If o_1 is a variable then we build the vector δ_j with all values equal to 1. Then if s_1 and/or p_1 are constant, we have to filter r with the map function between properties of $\mathcal{G}$ and a suitable space $\mathbb{F}$, *e.g.*, natural numbers for a boolean result. Therefore

$$\widetilde{r} = \mathrm{map}(p(\cdot) == p_1 \wedge i == \mathrm{idx}_0(s_1), r) \,.$$

The reader should notice a shorthand notation for an implicit boolean function: as for all descendant of the C programming language, such definition yields 1 if the condition is met, 0 otherwise.

If we process t_1 by the object, given $i = \mathrm{idx}_1(s_1)$, we build a Kroneker vector as a vector δ_i, with $|\delta_i| = |\mathcal{S}|$, and finally apply of the rank-2 tensor represented by M_{ij} to δ_i, *i.e.*: $r = M_{ij}\delta_i$. If s_1 is a variable then we build the vector δ_i with all values equal to 1. Similarly to the processing by subject, if o_1 and/or p_1 are constant, we have to filter r with the map function: $\widetilde{r} = \mathrm{map}(p(\cdot) == p_1 \wedge j == \mathrm{idx}_1(o_1), r)$. In the following, given a triple-pattern t, we will denote the processing by subject and the processing by object with $\mathcal{F}_s(t)$ and $\mathcal{F}_o(t)$, respectively.

For instance, referring to the example pictured in Figure 1, let us consider the triple pattern $t_1 = \langle a, ?x, b \rangle$. If we process t_1 by subject, we have $j = \mathrm{idx}_1(b) = 2$ and $\delta_{j=2} = \{\{2\} \to 1\}$. Therefore we result

$$\mathcal{F}_s(t_1) = \mathrm{map}(i == \mathrm{idx}_0(a), M_{ij}\delta_j) = \{1, 2\} \to (p_1) \,.$$

The result $\{1, 2\} \to (p1)$ corresponds to the triple $\langle a, p_1, b \rangle$. Analogously, if we process t_1 by object, we have $i = \mathrm{idx}_0(a) = 1$ and $\delta_{i=1} = \{\{1\} \to 1\}$. Therefore we result

$$\mathcal{F}_o(t_1) = \mathrm{map}(j == \mathrm{idx}_1(b), M_{ij}\delta_i) = \{1, 2\} \to (p_1) \,.$$

We highlight the processing $\mathcal{F}$ instantiates the triple patterns, *i.e.* $\mathcal{F}_s(t_1) = \{1, 2\} \to (p_1)$ means that the triple $\langle a, p_1, b \rangle$ is an instance of the triple pattern t_1.

Then, an important task is to process a graph pattern with two (or more) triple patterns in a SPARQL query. Let us consider two triple patterns $t_1 = \langle s_1, p_1, o_1 \rangle$ and $t_2 = \langle s_2, p_2, o_2 \rangle$, such that $t_1 \leftrightarrow t_2$. Now we process each pattern by the role (*i.e.* subject or object) of the node that

is in common. Then given $\widetilde{r_1}$ and $\widetilde{r_2}$ the results from the processing of t_1 and t_2, respectively, we combine them by executing the Hadamard product: $r = \widetilde{r_1} \circ \widetilde{r_2}$. For instance, referring to the example pictured in Figure 1, let us consider the triple patterns $t_1 = \langle a, ?x, ?y\rangle$ and $t_2 = \langle ?y, ?z, c\rangle$. Such patterns have the variable $?y$ in common, that is object in t_1 and subject in t_2, *i.e.* we have to process t_1 by object and t_2 by subject. In this case we have

$$\widetilde{r_1} = \mathcal{F}_o(t_1) = M_{ij}\delta_{i=\mathrm{idx}_0(a)} = \{\{1,2\} \rightarrow (p1), \{1,4\} \rightarrow (p_2)\}\,.$$

$$\widetilde{r_2} = \mathcal{F}_s(t_2) = M_{ij}\delta_{j=\mathrm{idx}_1(c)} = \{\{2,3\} \rightarrow (p_3)\}.$$

Finally $r = \widetilde{r_1} \circ \widetilde{r_2} = \{\{1,2\} \rightarrow (p_1), \{2,3\} \rightarrow (p_3)\}$. In particular we organize the result r as a map, $\langle key\text{-}value\rangle$, where the key is a triple pattern and the value is the set of corresponding instances. In the example we have $r = \{\langle t_1, \{\{1,2\} \rightarrow (p_1)\}\rangle, \langle t_2, \{\{2,3\} \rightarrow (p_3)\}\rangle\}$.

Now let us consider a graph pattern with three triple patterns $t_1 = \langle s_1, p_1, o_1\rangle$, $t_2 = \langle s_2, p_2, o_2\rangle$ and $t_3 = \langle s_3, p_3, o_3\rangle$, such that $t_1 \leftrightarrow t_2$ and $t_2 \leftrightarrow t_3$. Therefore we have to process t_1 with t_2 and t_2 with t_3. In this case t_2 may be processed in two different ways, *i.e.* both by subject and by object. Then we calculate $r' = \widetilde{r_1} \circ \widetilde{r_2}' = \{\langle t_1, I_{t_1}\rangle, \langle t_2, I'_{t_2}\rangle\}$ and $r'' = \widetilde{r_2}'' \circ \widetilde{r_3} = \{\langle t_2, I''_{t_2}\rangle, \langle t_3, I_{t_3}\rangle\}$, where $\widetilde{r_1}$ and $\widetilde{r_3}$ come from processing t_1 and t_3, respectively, while $\widetilde{r_2}'$ and $\widetilde{r_2}''$ come from processing t_2 in two different ways. Finally to perform the graph pattern, composed by t_1, t_2 and t_3, we execute a *composition*, denoted by $\oplus$, that is $r = r' \oplus r'' = \{\langle t_1, I_{t_1}\rangle, \langle t_2, \{I'_{t_2} \cap I''_{t_2}\}\rangle, \langle t_3, I_{t_3}\rangle\}$. Such operation executes the union between the two processing but filtering r'_2 and r''_2 from all instances of t_2 that do not satisfy both r'_2 and r''_2 (*i.e.* we keep the intersection between the instances of t_2). If t_2 is processed in the same way as with t_1 as with t_2, then $\widetilde{r_2} = \widetilde{r_2}' = \widetilde{r_2}''$, *i.e.* $r = \widetilde{r_1} \circ \widetilde{r_2} \circ \widetilde{r_3}$.

For instance, let us refer again to Figure 1. Consider three triple patterns $t_1 = \langle a, ?y, ?x\rangle$, $t_2 = \langle c, ?z, ?x\rangle$ and $t_3 = \langle c, ?p, f\rangle$, *i.e.* $t_1 \leftrightarrow t_2$ and $t_2 \leftrightarrow t_3$. In this case t_2 has to be processed by object due to the intersection with t_1 and by subject due to the intersection with t_3. In this case we have

$$\left[\begin{array}{c|l}
\mathcal{F}_o(t_1) \circ \mathcal{F}_o(t_2): & \begin{array}{lcccccc} t_1: & 0 & 0 & 0 & p_2 & 0 & 0 \\ t_2: & 0 & 0 & 0 & p_5 & 0 & 0 \end{array} \\
\hline
\mathcal{F}_s(t_2) \circ \mathcal{F}_s(t_3): & \begin{array}{lcccccc} t_2: & (0 & 0 & p_5 & 0 & 0 & 0)^t \\ & (0 & 0 & p_6 & 0 & 0 & 0)^t \\ & (0 & 0 & p_4 & 0 & 0 & 0)^t \\ \hline t_3: & (0 & 0 & p_4 & 0 & 0 & 0)^t \end{array}
\end{array}\right]$$

The execution of the graph pattern $\{t_1, t_2, t_3\}$ provides

$$r = \{\mathcal{F}_o(t_1) \circ \mathcal{F}_o(t_2)\} \oplus \{\mathcal{F}_s(t_2) \circ \mathcal{F}_s(t_3)\}$$

where $I'_{t_2} \cap I''_{t_2} = \{3,4\} \rightarrow (p_5)$, and

$$r = \{\langle t_1, \{\{1,4\} \rightarrow p_2\}\rangle, \langle t_2, \{\{3,4\} \rightarrow (p_5)\}\rangle, \langle t_3, \{\{3,6\} \rightarrow (p_4)\}\rangle\}$$

It is straightforward to process graph patterns with more then three triple patterns, performing Hadarmard products and compositions, as we will show in the next paragraph. As final remark, we discuss the maintenance of such framework. In order to eliminate duplicates (or erroneous data) or obsolete information, or to update the RDF dataset with new data (as it is the nature of Web of data), state-of-the-art approaches have to provide insertion and deletion operations, often complex tasks to maintain the organization of the RDF dataset (*e.g.* see [22]). As a matter of fact, in our computational framework these procedures are reflected by *assigning*

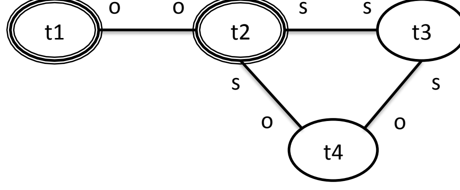


Figure 2: An example of Execution Graph

values in a given tensor. Removing a value is comfortably implemented by assigning the null value to an element, *i.e.*, $M_{ij} = (0)$, and analogously, inserting or modifying a value is attained via a simple operation $M_{ij} = (p)$. Similarly if we delete or insert a node, we add or remove a row or a column.

5.3 SPARQL execution

Exploiting the tensor applications defined above, we can execute a SPARQL query Q in terms of Hadarmard products and compositions. In our framework we have to define an *Execution Graph* E_G, supporting the automatic translation of Q into tensor applications and then the execution of such applications on our model. Formally we define E_G as follow

Definition 7 (Execution Graph) *Given a query $Q = \langle RC, G_P \rangle$, an Execution Graph E_G is a 3-tuple $\langle V, E, H \rangle$, where V is a set of nodes, that are triple patterns in G_P, E is a set of edges of the form (t_1^i, t_2^j) where $t_1, t_2 \in V$, $i, j \in \{s, o\}$ and H is a map $\langle key, value \rangle$ where the key is a variable in RC and the value is a set of triple patterns containing the key. In E_G an edge (t_1^i, t_2^j) means that $t_1 \leftrightarrow t_2$ and i, j indicate how to process t_1 and t_2,* i.e.*, by subject (s), or by object (o).*

For instance, referring to Example 1, let us consider the query

$$\texttt{Q1} = \{RC = \{?x\}, G_P = \langle \mathcal{T} = \{\{t_1, t_2, t_3, t_4\}\}, _, _, _ \rangle\}$$

where $t_1 = \langle a, ?y, ?x \rangle$, $t_2 = \langle c, ?z, ?x \rangle$, $t_3 = \langle c, ?p, f \rangle$ and $t_4 = \langle b, ?w, c \rangle$. The corresponding E_G is depicted in Figure 2. The nodes of E_G are t_1, t_2, t_3 and t_4, the edges are (t_1^o, t_2^o), (t_2^s, t_3^s), (t_2^s, t_4^o) and (t_3^s, t_4^o), and the set H is $\{\langle ?x, \{t_1, t_2\} \rangle\}$. In the graph, the double marked line indicates that the nodes contain a variable in H, while each edge depicts closed to the connected node how it will be processed. In this example `Q1` presents G_P with $\mathcal{T}$ containing a single set T of triple patterns. If $\mathcal{T}$ provides multiple disjunctive sets T_i, we will produce an execution graph for each T_i.

Algorithm 1 describes the function `BuildGraph` to build an execution graph E_G from a SPARQL query Q. In this case we assume $\mathcal{T}$ as a single set T of triple patterns (otherwise we have to perform the algorithm more times for each T_i). The set of nodes V is the copy of the set T from G_P (line [1]). The set H is built by using the function `Create_H` that takes as input the set of selection variables and the triple patterns of T (we omit the pseudocode since it is straightforward). Then the algorithm iterates on T (lines [4-6]) and extracts a triple pattern `tp` until T is empty. For each `tp` it invokes a recursive function `FindCC_AUX` (line [6]) that generates the edges to insert into E by computing the connected component between `tp` and the triples patterns in T. As shown in Algorithm 2, `FindCC_AUX` searches all triple patterns

Algorithm 1: Building of the Execution Graph

Input : Result clause RC and triple patterns set T
Output: The corresponding Execution Graph E_G

```
1  V ← T;
2  H ← Create_H(RC,T);
3  E ← ∅;
4  while T is not empty do
5  |   T ← T ∖ {tp};
6  |_  E ∪ FindCC_AUX(tp, T )
7  return E_G : ⟨V, E, H⟩;
```

tp_i such that tp_i and `tp` present an intersection (i.e. $tp_i \leftrightarrow$ `tp`). If there exists tp_i, then the algorithm extract it from T (line [3]) and calculates for both tp_i and `tp` which role the node in common presents, *i.e.* subject or object (lines [4-13]). Then we can add the new edge to the set `PQ` representing the connected component and we recall the recursion (line [14]).

Algorithm 2: FindCC_AUX: a recursive function supporting the building of E_G

Input : A triple pattern tp, a graph pattern T.
Output: The connected component built from tp.

```
1  PQ ← ∅;
2  foreach tp_i ∈ T: tp_i ↔ tp do
3  |   T ← T − tp_i;
4  |   if tp_i.s == tp.s then
5  |   |_  PQ ← PQ ∪ {tp^s, tp_i^s};
6  |   else
7  |   |   if tp_i.s == tp.o then
8  |   |   |_  PQ ← PQ ∪ {tp^s, tp_i^o};
9  |   |   else
10 |   |   |   if tp_i.o == tp.s then
11 |   |   |   |_  PQ ← PQ ∪ {tp^o, tp_i^s};
12 |   |   |   else
13 |   |   |_  |_  PQ ← PQ ∪ {tp^o, tp_i^o};
14 |_  PQ ← PQ ∪ FindCC_AUX(tp_i, T );
15 return PQ ;
```

Once generated the execution graph E_G, to execute the corresponding SPARQL query, we simply explore E_G by using a *Depth-First Search* (DFS) traversal. First of all we execute the triple patterns in $\mathcal{T}$ of G_P, as described by Algorithm 3. The algorithm traverses E_G to group all processing of triple patterns to execute as Hadamard products. Finally we perform the composition of all groups. The algorithm starts from an empty set HP, representing the set of all groups CL. Then it iterates on the set of edges E and extracts (casually) the first element t_i^x of a pair $(t_i^x, t_z^w) \in E$ (line [3]). Such operation is performed by the function `take` (we

Algorithm 3: Execution of the graph pattern

Input : An Execution Graph E_G
Output: The result from the query answering

```
1 HP ← ∅;
2 while E is not empty do
3 |   t_i^x ← take(E);
4 |   CL ← {F_x(t_i)};
5 |   HP ∪ {CreateCluster(E, t_i^x, CL)};
6 return Composition (HP);
```

omit the pseudo code since it is straightforward). We initialize a group CL with the processing $\mathcal{F}_x(t_i)$ (line [4]) and invoke a recursive function `CreateCluster` to fill CL by traversing E_G in depth (line [5]). Each time `CreateCluster` traverses an edge, it removes that edge from E. Therefore the iteration will finish when E will be empty. At the end the function `Composition` executes the compositions ($\oplus$) between the multiple sets of HP. The function converts each set $CL_i \in$ HP in a map structure $\langle T, I \rangle$, where the keys T are the triple patterns while the values I are the corresponding instances. Then for each key t_j occurring in more than one cluster in HP, the function calculates the intersection between each value I_j and updates the value of the maps identified by the key t_j with the intersection. (*i.e.* for space constraints we do not include the pseudo code of the function, that is very simple). Algorithm 4 illustrates the code of `CreateCluster`. If there does not exist a pair $(t_i^x, t_j^y) \in E$ then the traversing is

Algorithm 4: Create the group of Hadamard products

Input : A set of edges E of E_G, a triple pattern t_i^x, a group CL
Output: The updated Group CL

```
1 if ∄(t_i^x, t_j^y) ∈ E then
2 |   return CL;
3 else
4 |   foreach (t_i^x, t_j^y) ∈ E do
5 |   |   CL ← CL ∘ {F_y(t_j)};
6 |   |   E ∖ {(t_i^x, t_j^y)};
7 |   |   CreateCluster(E, t_j^y, CL);
```

finished and we return the group CL (lines [1-2]). Otherwise, for each (t_i^x, t_j^y) we update CL adding the processing $\mathcal{F}_y(t_j)$ (line [5]), we remove the pair from E (line [6]) and we recall the recursion on t_j^y (line [7]).

For instance, referring to the execution graph of Figure 2, we obtain the following processing

$$\left[\begin{array}{lllllll}\mathcal{F}_o(t_1): & 0 & p_1 & 0 & p_2 & 0 & 0 \\ \mathcal{F}_o(t_2): & 0 & 0 & 0 & p_5 & p_6 & p_4 \\ \mathcal{F}_s(t_2): & (0 & 0 & p_5 & 0 & 0 & 0)^t \\ & (0 & 0 & p_6 & 0 & 0 & 0)^t \\ & (0 & 0 & p_4 & 0 & 0 & 0)^t \\ \mathcal{F}_s(t_3): & (0 & 0 & p_4 & 0 & 0 & 0)^t \\ \mathcal{F}_o(t_4): & 0 & 0 & p_3 & 0 & 0 & 0\end{array}\right]$$

Performing the Algorithm 3, we obtain two groups: $CL_1 = \{\mathcal{F}_o(t_1) \circ \mathcal{F}_o(t_2)\}$ and $CL_2 = \{\mathcal{F}_s(t_2) \circ \mathcal{F}_s(t_3) \circ \mathcal{F}_o(t_4)\}$ as follows

$$\left[\begin{array}{l|lllllll}\mathcal{F}_o(t_1) \circ \mathcal{F}_o(t_2): & t_1: & 0 & 0 & 0 & p_2 & 0 & 0 \\ & t_2: & 0 & 0 & 0 & p_5 & 0 & 0 \\ \hline \mathcal{F}_s(t_2) \circ \mathcal{F}_s(t_3) \circ \mathcal{F}_o(t_4): & t_2: & (0 & 0 & p_5 & 0 & 0 & 0)^t \\ & & (0 & 0 & p_6 & 0 & 0 & 0)^t \\ & & (0 & 0 & p_4 & 0 & 0 & 0)^t \\ \cline{2-8} & t_3: & (0 & 0 & p_4 & 0 & 0 & 0)^t \\ \cline{2-8} & t_4: & 0 & 0 & p_3 & 0 & 0 & 0\end{array}\right]$$

The composition of this two groups results four elements: $\{1,4\} \rightarrow (p_2)$, $\{3,4\} \rightarrow (p_5)$, $\{3,6\} \rightarrow (p_4)$ and $\{2,3\} \rightarrow (p_3)$. They represent instances of the corresponding triple patterns, as follows

I_{t_1}	I_{t_2}	I_{t_3}	I_{t_4}
$\langle a, p_2, d\rangle$	$\langle c, p_5, d\rangle$	$\langle c, p_4, f\rangle$	$\langle b, p_3, c\rangle$

Now to filter the result following the *result clause*, we use H to extract all triple patterns where the variables in H occur. Then using the values coming from the processing of T we instantiate the variables thought the key-value links of H. In our example $RC = \{?x\}$. The set H indicates t_1 and t_2 linked to $?x$. Therefore since $?x$ is object, and the composition returned $\{3,4\} \rightarrow (p_5)$ for t_2, the value 4 is the instance of $?x$, *i.e.* it corresponds to the node d.

Through our model, the processing of the operators $\{$FILTER, OPTIONAL, UNION$\}$ is quite simple. Let us consider the query

$$\begin{aligned}\texttt{Q2} = \{\ \ & RC = \{?y, ?w\}, \\ & G_P = \langle \mathcal{T} = \{\{t_1, t_2\}\}, f = \{?y! = a\}, OPT = \{t_3\}, _\rangle\}\end{aligned}$$

from Example 1. In the query we have three triple patterns $t_1 = \langle a, ?z, ?x\rangle$, $t_2 = \langle ?y, ?p, ?x\rangle$ and $t_3 = \langle ?x, ?w, e\rangle$, where t_3 is optional. In this case we build two execution graphs E'_G and E''_G where E'_G is built on t_1 and t_2 while E''_G is built on all the patterns t_1, t_2 and t_3. The idea is to execute two queries with and without t_3 and then to make the union of the results. Generalizing, given the set T of triple patterns and $OPT = \{opt_1, opt_2, \ldots, opt_m\}$ from a query Q we build $m + 1$ execution graphs, that is E_G on T and m execution graphs E^i_G on T and t_{opt_i} (*i.e.* t_{opt_i} is the triple pattern from opt_i). Recalling $\texttt{Q2}$, we have $r_1 = \mathcal{F}_o(t_1) \circ \mathcal{F}_o(t_2)$ from E'_G and $r_2 = \mathcal{F}_o(t_1) \circ \mathcal{F}_o(t_2) \circ \mathcal{F}_s(t_3)$ from E''_G, resulting as follows

	I_{t_1}	I_{t_2}	I_{t_3}
r_1 :	$\langle a, p_2, d\rangle$	$\langle a, p_2, d\rangle$ $\langle c, p_5, d\rangle$	
r_2 :	$\langle a, p_2, d\rangle$	$\langle a, p_2, d\rangle$ $\langle c, p_5, d\rangle$	$\langle d, p_7, e\rangle$

In this case the final result r is r_2 itself (*i.e.* r_1 is subsumed by r_2). Finally we have to apply the FILTER constraint to r before processing the *result clause*. In fact, the instance $\langle a, p_2, d\rangle$ of t_2 does not satisfy the boolean constraint $?y$!= a. Supported by the set H of the execution graph we select the instances of the triple patterns where the variables in the filter constraint occur, *i.e.* I_{t_2} = $\{\{1,4\} \to (p_2), \{3,4\} \to (p_5)\}$, and we apply the boolean condition through the map function. In the example, given $\mathrm{idx}_0(a)$ = 1, we have $\mathrm{map}(i$!= $1, I_{t_2})$, that is to delete $\{1,4\} \to (p_2)$ from I_{t_2}. Finally applying the *result clause*, the variables $?y$ and $?w$ are instantiated by the values 4 (*i.e.* d) and p_7.

The UNION operator is processed quite similar to OPTIONAL in our model. Referring to Example 1, let us consider the query

$$\begin{aligned} \mathtt{Q3} = \{ \quad & RC = \{*\}, \\ & G_P = \langle \mathcal{T} = \{T_1 = \{t_1\}, T_2 = \{t_2\}\}, _, _, U = \{(T_1, T_2)\}\rangle\} \end{aligned}$$

where t_1 = $\langle ?a, ?z?x\rangle$ and t_2 = $\langle c, ?p?y\rangle$. In this case for each $T_i \in \mathcal{T}$ we build an execution graph E^i_G, we perform E^i_G to return r_i and finally we simply make the union of each r_i corresponding to each $T_i \in U$. In our example, first of all, we have to process t_1 and t_2 separately. In this case, since each E^i_G has no edges and only one node, we process t_1 and t_2 indifferently by subject or by object. Therefore we have $\mathcal{F}_o(t_1)$ and $\mathcal{F}_o(t_2)$ resulting the following instances

I_{t_1}	I_{t_2}
$\langle a, p_1, b\rangle$ $\langle a, p_2, d\rangle$	$\langle c, p_5, d\rangle$ $\langle c, p_6, e\rangle$ $\langle c, p_4, f\rangle$

The final result is the union $I_{t_1} \cup I_{t_2}$.

6 Experimental Results

We implemented our framework into TENSORRDF, a Java system for SPARQL answering over RDF datasets. All procedures for processing and maintenance the RDF dataset are linked to the Mathematica 8.0[12] computational environment (*i.e. Jlink*[13]). In this section we will show how much TENSORRDF improves query performance with respect to other systems. Experiments were conducted on a quad core 2.66GHz Intel Xeon, with 12 GB of memory, 6 MB cache, and a 4-disk 1Tbyte striped RAID array, running RedHat Linux and Java 1.6 installed.

Benchmark Environment. We evaluated the performance of our system comparing with the triple stores Virtuoso, Sesame, Jena-TDB, and BigOWLIM (see Section 2 for references), and the the open-source system RDF-3X. The configuration and the version of each system follows:

[12] `http://www.wolfram.com/mathematica/`

[13] `http://reference.wolfram.com/mathematica/JLink/tutorial/Overview.html`

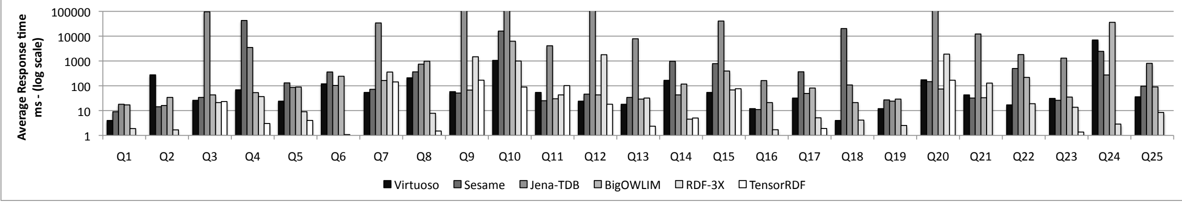


Figure 3: Response Times on DBpedia: bars refer each system, using different gray scales, *i.e.* from Virtuoso, black bar, to TENSORRDF, light gray

- *Virtuoso Open-Source Edition version 6.1.2*. We set the following memory-related parameters: `NumberOfBuffers` = 1048576, `MaxDirtyBuffers` = 786432.
- *Sesame Version 2.3.2* with Tomcat 6.0 as HTTP interface. We used the native storage layout and set the $spoc$, $posc$, $opsc$ indices in the native storage configuration. We set the Java heap size to 4GB.
- *Jena-TDB Version 0.8.7* with Joseki 3.4.3 as HTTP interface. We configured the TDB optimizer to use statistics. This mode is most commonly employed for the TDB optimizer, whereas the other modes are mainly used for investigating the optimizer strategy. We also set the Java heap size to 4GB.
- *BigOWLIM Version 3.4*, with Tomcat 6.0 as HTTP interface. We set the entity index size to 45,000,000 and enabled the predicate list. The rule set is empty. We set the Java heap size to 4GB.
- *RDF-3X Version 0.3.5*, including the scalable join processing approach discussed in [21].

In our experiments, we employed three datasets: DBPEDIA (version 3.6), *i.e.* 200 M triples loaded into the official SPARQL endpoint[14], UNIPROT[15], *i.e.* 500 M triples consisting of 57GB of protein information, and BILLION, the dataset of Billion Triples Challenge[16] that is 1000 M triples. For each dataset we defined a set of test queries. For DBPEDIA we wrote 25 queries of increasing complexity, as shown in the Appendix A. Such queries involve SELECT SPARQL queries embedding *concatenation* (.), FILTER, OPTIONAL and UNION operators. Table 1 illustrates the features of each query. In particular, we show the number of triple patterns (*i.e.* $|G_P|$), and which construct is involved (*i.e.* the sign X fills the corresponding cell). Referring to UNIPROT and BILLION, we exploit the test queries defined in [21], that are 8 SELECT SPARQL queries for each dataset, involving only the *concatenation* (.).

Performance. First of all we would show the *Importing Performance* of our system. Table 2 illustrates the times (seconds) and the memory usage (MB) to import and create the tensor for each dataset. In this case we used the Mathematica framework to create the tensor in memory on centralized massive analysis. As shown by the Table, the importing times are significant (*i.e.* dozens of seconds) to import in our framework huge triples volumes. Of course the memory usage is a relevant cost: for 1000 M triples we need 1760 MB, that is supported by free-memory

[14]`http://dbpedia.org/sparql`
[15]`http://dev.isb-sib.ch/projects/uniprot-rdf/`
[16]Semantic Web Challenge 2008. Billion triples track. `http://challenge.semanticweb.org/`

Table 1: Features of DBpedia queries

Query	$\|G_P\|$	Filter	Optional	Union
Q1	1			
Q2	1	X		
Q3	4		X	
Q4	2			X
Q5	5			X
Q6	3	X		
Q7	11		X	
Q8	3	X		X
Q9	7		X	X
Q10	8	X	X	
Q11	9		X	X
Q12	3			
Q13	2	X		
Q14	3			X
Q15	4		X	
Q16	3		X	
Q17	3			X
Q18	2	X		X
Q19	2	X		X
Q20	10	X	X	X
Q21	1			
Q22	2			
Q23	2			
Q24	4			
Q25	4			

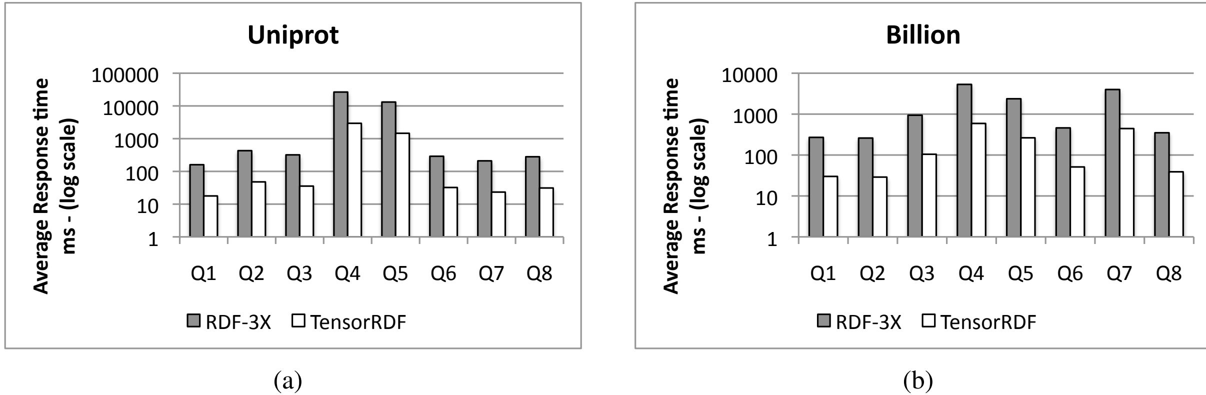
constraint environments. Nevertheless 200 M triples require 350 MB that is a reasonable quantity of memory supported by modern high-technology mobile devices. As a side-note, we highlight the fact that we could make division in smaller blocks, *e.g.* 200 M triples, and performs analysis in a distributed environment (*i.e.* Grid).

Then the most important experiment is the query execution on each dataset, comparing the performance of each system. We ran the queries ten times and measured the average response time, *i.e.* ms and logarithmic scale. We performed *cold-cache* experiments (by dropping all file-system caches before restarting the systems and running the queries). Figure 3 illustrates the response times on DBPEDIA. On the average Sesame and Jena-TDB perform poorly, BigOWLIM and Virtuoso better, and RDF-3X is really competitive. TENSORRDF outperforms all competitors, in particular also RDF-3X by a large margin. We computed the speed-up for all queries as the ratio between the execution time of RDF-3X, and that of our approach, or briefly $S = t_{RDF-3X}/t_{\text{TENSORRDF}}$. The query performance of TENSORRDF is on the average 18 times better than that of RDF-3X, 128 times on the maximum, *i.e.*, Q21. In particular the queries involving OPTIONAL and UNION operators require the most complex computation: triple stores, *i.e.* Virtuoso, BigOWLIM, Sesame and Jena-TDB, depends on the physical organization of indexes, not always matching the joins between patterns. RDF-3X provides a permutation of all com-

Table 2: Importing Performance

	#triples	Time to Import	Memory Usage
DBPEDIA	200 M	45 sec	350 MB
UNIPROT	500 M	110 sec	880 MB
BILLION	1000 M	230 sec	1760 MB

binations of indexes on subject, property and object of a triple to improve efficiency. However queries, embedding OPTIONAL and UNION operators in a graph pattern with a considerable size, require complex joins between huge number of triples (*i.e.* Q20) that compromises the performance. Due to the *sparse matrix* representation of tensors and vectors, TENSORRDF exploits parallel computations to perform efficiently map functions, reductions, Hadamard products and compositions ($\oplus$).



(a) (b)

Figure 4: Response Times on UNIPROT (a), and BILLION (b)

Similarly, we executed the test queries on UNIPROT and BILLION. Since RDF-3X is the most performing with respect to the other competitors, we compared TENSORRDF directly with RDF-3X. We highlight that we used a set of SELECT SPARQL queries embedding only the *concatenation* (.) operator, on which RDF-3X exploits its physical indexing in a profitable way. Figure 4 presents the results. Also in this case TENSORRDF outperforms RDF-3X by a large margin, due to the fact our system has to perform only Hadamard products and compositions without any filtering or union operations.

Another strong point of our system is a very low consumption of memory for query execution, due to the *sparse matrix* representation of tensors and vectors. Figure 5 shows the memory usage (KB) to query DBPEDIA. On the average, all queries (also the most complex) require very few bytes of memory (*i.e.* dozens of KBytes). Querying UNIPROT and BILLION presents a similar behavior (*i.e.* dozens of KBytes).

7 Conclusions and Future Work

We have presented an abstract algebraic framework for the efficient and effective analysis of RDF data. Our approach leverages tensorial calculus, proposing a general model that exhibits a great flexibility with queries, at diverse granularity and complexity levels (*i.e.* both *conjunctive* and *non-conjunctive* patterns with filters). Experimental results proved our method efficient

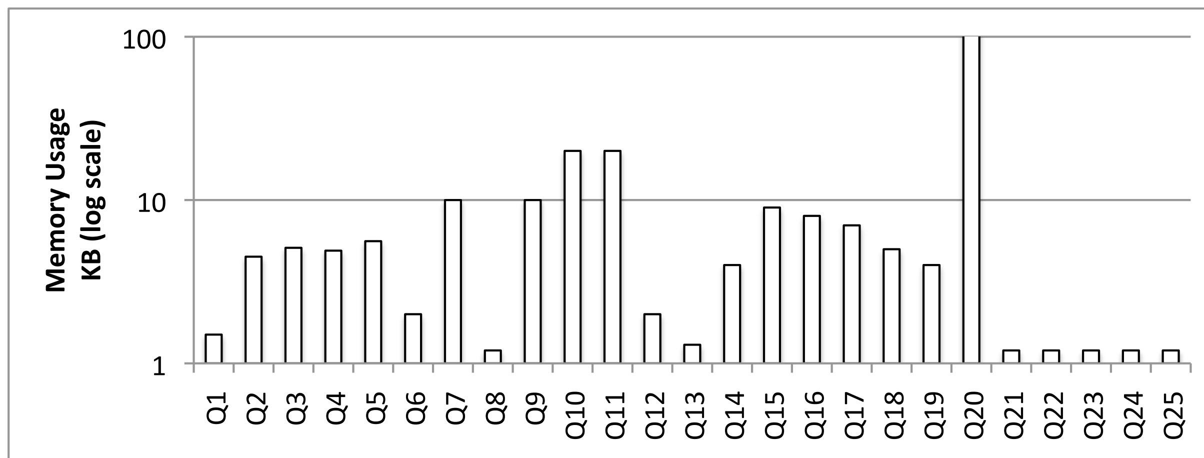


Figure 5: Memory Usage to query DBPEDIA

when compared to recent approaches, yielding the requested outcomes in memory constrained architectures. For future developments we are investigating the introduction of reasoning capabilities, along with a thorough deployment in highly distributed Grid environments. In addition, we are about to test our model on mobile devices, comprising more complex properties and queries.

References

[1] Daniel J. Abadi, Adam Marcus 0002, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[2] Ralph Abraham, Jerrold E. Marsden, and Tudor Ratiu. *Manifolds, Tensor Analysis, and Applications*. Springer, 1988.

[3] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In *Proc. of SemWeb*, pages 109–113, 2001.

[4] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world sparql queries. *CoRR*, abs/1103.5043, 2011.

[5] J. Beckmann, A. Halverson, R. Krishnamurthy, and J. Naughton. Extending RDBMSs to support sparse datasets using an interpreted attribute storage format. In *Proc. of ICDE*, pages 58–68, 2006.

[6] Adrian Bondy and U. S. R. Murty. *Graph Theory*. Graduate Texts in Mathematics. Springer, 2010.

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

[8] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient sql-based rdf querying scheme. In *VLDB*, pages 1216–1227, 2005.

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

[10] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[11] R. De Virgilio, F. Giunchiglia, and L. Tanca, editors. *Semantic Web Information Management - A Model-Based Perspective*. Springer, 2010.

[12] O. Erling and I. Mikhailov. RDF support in the virtuoso DBMS. In *Proc. of CSSW*, pages 7–24, 2007.

[13] George H. L. Fletcher and Peter W. Beck. Scalable indexing of rdf graphs for efficient join processing. In *CIKM*, pages 1513–1516, 2009.

[14] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4:250–269, 1978.

[15] John H. Heinbockel. *Introduction to Tensor Calculus and Continuum Mechanics*. Trafford Publishing, 2001.

[16] Kenneth M. Hoffman and Ray Kunze. *Linear Algebra*. Prentice Hall, 1971.

[17] B. Jancewicz. The extended grassmann algebra of $\mathbb{R}^3$. In William E. Baylis, editor, *Clifford (geometric) algebras with applications to physics, mathematics, and engineering*. Birkhäuser Boston, 1996.

[18] M. Hausenblas K Möller, R. Cyganiak, S. Handschuh, and G. Grimnes. Learning from linked open data usage: Patterns & metrics. In *Proceedings of the Web Science Conference 2010*, 2010.

[19] F. Manola and E. Miller. Resource description framework((RDF). World Wide Web Consortium, Recommendation, REC-rdf-primer-20040210, 2004.

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

[21] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD*, pages 627–640, 2009.

[22] Thomas Neumann and Gerhard Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases. *PVLDB*, 3(1):256–263, 2010.

[23] J Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[24] E Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. Technical report, W3C Recommendation, 2008.

[25] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In *ICDT*, pages 4–33, 2010.

[26] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. In *Proc. of VLDB*, pages 1553–1563, 2008.

[27] Lefteris Sidirourgos, Romulo Goncalves, Martin L. Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.

[28] Maria-Esther Vidal, Edna Ruckhaus, Tomas Lampo, Amadís Martínez, Javier Sierra, and Axel Polleres. Efficiently joining group patterns in sparql queries. In *ESWC*, pages 228–242, 2010.

[29] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[30] K. Wilkinson. Jena property table implementation. In *SSWS*, 2006.

A DBpedia Test Queries

```
dbpedia: http :// dbpedia . org / resource /
dbp-owl: http :// dbpedia . org / ontology /
dbp-prop: http :// dbpedia . org / property /
dbp-yago: http :// dbpedia . org / class / yago /
dbp-cat: http :// dbpedia . org / resource / Category /
rdf: http :// www .w3. org /1999/02/22 - rdf - syntax -ns#
rdfs: http :// www .w3.org /2000/01/ rdf - schema #
owl: http :// www .w3. org /2002/07/ owl #
xsd: http :// www .w3. org /2001/ XMLSchema #
skos: http :// www .w3.org /2004/02/ skos / core #
foaf: http :// xmlns . com/ foaf /0.1/
georss: http :// www. georss . org / georss /

Q1
SELECT DISTINCT ?v1
WHERE { ?x rdf:type ?v1. }

Q2
SELECT *
WHERE { ?x ?v2 ?v1.
             FILTER (? v2= dbpedia2:redirect || ?v2=dbp-prop:redirect ) }

Q3
SELECT ?v4 ?v8 ? v10
WHERE { ?v5 dbp-owl:thumbnail ?v4. ?v5 rdf:type dbp-owl:Person .
             ?v5 foaf:page ?v8.
             OPTIONAL { ?v5 foaf : homepage ?v10.}
             }

Q4
SELECT ?v6 ?v9 ?v8 ?v4
WHERE { { ?v6 foaf : name ?v8. }
              UNION { ?v9 foaf : name ?v4. }
            }

Q5
SELECT DISTINCT ?v3 ?v4 ?v5
WHERE { { ?v3 foaf:name ?v4; rdfs : comment ?v5. }
             UNION
             { ?v3 dbp-prop:series ?v8. ?v3 foaf : name ?v4; rdfs:comment ?v5. } }

Q6
SELECT DISTINCT ?v3 ?v5 ?v7
WHERE { ?v3 rdf:type dbp-yago:Company108058098 . ?v3 dbp - prop : numEmployees ?v5
```

```
            FILTER ( xsd:integer (?v5) >=10). ?v3 foaf:homepage ?v7. }

Q7
SELECT distinct ?v0 ?v1 ?v2 ?v3 ?v5 ?v6 ?v7 ? v10
WHERE { ?v0 rdfs : comment ?v1.
             OPTIONAL {? v0 skos : subject ?v6}
             OPTIONAL {? v0 dbp - prop :industry ?v5}
             OPTIONAL {? v0 dbp - prop : location ?v2}
             OPTIONAL {? v0 dbp - prop : locationCountry ?v3}
             OPTIONAL {?v0 dbp - prop : locationCity ?v9; dbp - prop : manufacturer ?v0}
             OPTIONAL {? v0 dbp - prop : products ? v11 ; dbp - prop :model ?v0}
             OPTIONAL {? v0 georss : point ? v10 } OPTIONAL {? v0 rdf : type ?v7 }}

Q8
SELECT ?v2 ?v4
WHERE { { ?v2 dbp - prop : population ?v4.
                FILTER ( xsd:integer (? v4) > 1000) }
                UNION { ?v2 rdf : type %% v1 %%. ?v2 dbp - prop :populationUrban ?v4.
                             FILTER (xsd: integer (? v4) > 500) } }

Q9
SELECT *
WHERE { ?v2 a dbp -owl: Settlement. ? v6 dbp -owl : city ?v2
              UNION {? v6 dbp - owl : location ?v2} {? v6 dbp - prop : iata ?v5 .}
              UNION {? v6 dbp - owl : iataLocationIdentifier ?v5.}
              OPTIONAL { ?v6 foaf : homepage ?v7. }
              OPTIONAL { ?v6 dbp - prop : nativename ?v8 .} }

Q10
SELECT DISTINCT ?v0
WHERE { ?v3 foaf : page ?v0. ?v3 rdf : type dbp - owl : SoccerPlayer . ?v3 dbp - prop : position ?v6 .
              ?v3 dbp - prop : clubs ?v8. ?v8 dbp - owl : capacity ?v1 .
              ?v3 dbp - owl : birthPlace ?v5 . ?v5 ?v4 ?v2.
              OPTIONAL {? v3 dbp - owl : number ?v9 .}
              FILTER (? v4 = dbp - prop : populationEstimate || ?v4 = dbp - prop : populationCensus
                        || ?v4 = dbp - prop : statPop )
              FILTER ( xsd : integer (? v2) > 11 ) .
              FILTER ( xsd : integer (? v9) < 11) .
              FILTER (? v6 = 'Goalkeeper '@en || ?v6 = dbpedia :Goalkeeper_ %28 association_football %29
                        || ?v6 = dbpedia : Goalkeeper_ %28 football %29) }

Q11
SELECT distinct ?v3 ?v4 ?v2
WHERE { {?x dbp - prop :subsid ?v3
              OPTIONAL {? v2 rdf:type dbp - prop : parent }
              OPTIONAL {?x dbp - prop : divisions ?v4 }}
              UNION {? v2 rdf:type dbp - prop : parent
                        OPTIONAL {?x dbp - prop : subsid ?v3}
                        OPTIONAL {?x dbp - prop : divisions ?v4 }}
                        UNION {?x dbp - prop : divisions ?v4
                        OPTIONAL {?x dbp - prop :subsid ?v3}
                        OPTIONAL {? v2 rdf:type dbp - prop : parent }} }

Q12
SELECT DISTINCT ?v5
WHERE { ?v2 rdf : type dbp - owl : Person . ?v2 dbp - owl: nationality ?v4 . ?v4 rdfs : label ?v5.}

Q13
SELECT DISTINCT ?v2 ?v3
WHERE { ?v2 rdf : type ?x; rdfs : label ?v3 .
              FILTER regex (?v3 , 'pes ', 'i') }

Q14
SELECT ?v0
WHERE {{ ?x rdfs : comment ?v0.}
              UNION {?x foaf : depiction ?v1}
              UNION {?x foaf : homepage ?v2 }}

Q15
SELECT ?v6 ?v8 ? v10 ?v4
WHERE { ?v4 skos : subject ?v5 . ?v4 foaf : name ?v6 .
```

```
              OPTIONAL { ?v4 rdfs : comment ?v8 .}
              OPTIONAL { ?v4 rdfs : comment ? v10 . } }

Q16
SELECT DISTINCT ?x ?v6 ?v7
WHERE { ?x ?v4 ?v5.
             OPTIONAL {? v5 rdfs : label ?v6} .
             OPTIONAL {? v4 rdfs : label ?v7 }}

Q17
SELECT DISTINCT ?v2 ?x
WHERE{ {? v2 skos : subject ?x}
             UNION {? v2 skos : subject dbp - cat: Prefectures_in_France.}
             UNION {? v2 skos : subject dbp - cat :German_state_capitals .}
             }

Q18
SELECT ?v3 ?v4 ?v5
WHERE { { ?x ?v3 ?v4.
             FILTER (( STR (? v3) = 'http :// www .w3. org /2000/01/ rdf - schema #label '
                            && lang (? v4) = 'en ')
                            || ( STR (? v3) = 'http ://dbpedia . org / ontology / abstract '
                            && lang (? v4) = 'en ')
                            || ( STR (? v3) = 'http :// www .w3.org /2000/01/ rdf - schema #comment '
                            && lang (? v4) = 'en ')
                            || ( STR (? v3) != 'http:// dbpedia .org/ ontology / abstract '
                            && STR (? v3) != 'http :// www .w3. org /2000/01/ rdf - schema # comment '
                            && STR(? v3) != 'http :// www .w3. org /2000/01/ rdf - schema # label ') ) }
             UNION { ?v5 ?v3 ?y FILTER ( STR (? v3) = 'http:// dbpedia .org/ ontology / owner '
                            || STR (? v3) = 'http ://dbpedia . org / property / redirect ' ) } }

Q19
SELECT ?v1
WHERE { { ?v1 rdfs : label ?x }
             UNION { ?v1 rdfs : label ?y }.
             FILTER ( regex ( str (? v1),'http ://dbpedia . org / resource /')
             || regex ( str (? v1),'http ://dbpedia . org / ontology /')
             || regex ( str (? v1),'http :// www.w3.org /2002/07/ owl ')
             || regex ( str (? v1),'http :// www .w3.org /2001/ XMLSchema ')
             || regex ( str (? v1),'http :// www.w3.org /2000/01/ rdf - schema ')
             || regex ( str (? v1),'http:// www .w3. org /1999/02/22 - rdf -syntax -ns ')) }

Q20
SELECT *
WHERE { ?v6 a dbp -owl: PopulatedPlace ; dbp -owl : abstract ?v1; rdfs : label ?v2;
            geo :lat ?v3; geo :long ?v4.
            UNION { ?v5 dbp -prop : redirect ?v6. }
            OPTIONAL { ?v6 foaf : depiction ?v8 }
            OPTIONAL { ?v6 foaf : homepage ?v10 }
            OPTIONAL { ?v6 dbp - owl : populationTotal ? v12 }
            OPTIONAL { ?v6 dbp - owl: thumbnail ? v14 }
            FILTER ( xsd : integer (? v12) > 11) . }

Q21
SELECT * WHERE { ?x dbp - prop : redirect ?v0 . }

Q22
SELECT ?v2 WHERE { ?v3 foaf : homepage ?v2 . ?v3 rdf :type ?x . }

Q23
SELECT ?v4 WHERE { ?v2 rdf : type dbp -owl: Person . ?v2 foaf : page ?v4 . }

Q24
SELECT *
WHERE { ?v1 a dbp -owl: Organisation . ?v2 dbp- owl : foundationPlace ?y .
             ?v4 dbp - owl : developer ?v2 . ?v4 rdf:type ?x . }

Q25
SELECT ?v0 ?v1 ?v2 ?v3
WHERE { ?v6 dbp - prop : name ?v0. ?v6 dbp - prop : pages ?v1.
                 ?v6 dbp- prop : isbn ?v2. ?v6 dbp - prop : author ?v3 .}
```