Distributed in-memory SPARQL Processing via DOF Analysis

Roberto De Virgilio
Roma Tre University, Rome, Italy
dvr@dia.uniroma3.it

ABSTRACT

Using so-called *triple patterns* as building blocks, SPARQL queries search for specified patterns in 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 DBMS (or disk) based solutions, with high consumption of resources; moreover, these exhibit very limited flexibility dealing with queries, at various levels of granularity and complexity (*e.g.*, SPARQL queries involving UNION or OPTIONAL operators). In this paper we propose a computational in-memory framework for distributed SPARQL query answering, based on the notion of *degree of freedom* of a triple. This algorithm relies on a general model of RDF graph based on the first principles of linear algebra, in particular on *tensorial calculus*. Experimental results show that our approach, utilizing linear algebra techniques can process analysis efficiently, when compared to recent approaches.

CCS Concepts

•Information systems → Data management systems; World Wide Web; •Computing methodologies → Linear algebra algorithms; Distributed algorithms;

Keywords

RDF; SPARQL; Tensor Calculus

1. INTRODUCTION

Today, many organizations and practitioners are all contributing to the "Web of Data", building RDF repositories of huge amounts of semantic data, posing serious challenges in maintaining and querying large datasets. Modern scenarios involve analyses of very large semantic datasets, usually employing the SPARQL query language. Many aspects of large-scale RDF data management have already been studied in the database research community, including native RDF storage layout and index structures [18], SPARQL query processing and optimization [8], as well as formal semantics and computational complexity of SPARQL [20, 23].

Challenges. Examining the prevalent trend in semantic information storage and inspection, we face two major challenges: management of large datasets, and scalability of both storage and querying. As size increases, both storage and analysis must scale accordingly; however, despite major efforts, building performant and scalable RDF systems is still a hurdle. Most of the current state-of-the-art approaches consider SPARQL as *the SQL for RDF*, and therefore they usually employ RDBMS-based solutions to store RDF graphs, and to execute a SPARQL query through SQL engines (e.g. Jena or Sesame). Moreover, popular systems are developed as single-machine tools [18, 28], which hinder performances as the size of RDF dataset continues to escalate. In particular Jena, Sesame, RDF-3X [18], BitMat [1], TripleBit [29] and GADDI [31] represent centralized approaches exploiting single-machine implementations of edge (e.g., [18, 1]) and subgraph (e.g. [31]) index based approaches to graph matching over graph-shaped data (e.g. RDF). In this context efficiency is driven by various forms of horizontal and vertical partitioning scheme [4], or by sophisticated encoding schemes [1, 29]. Hence, such proposals require the replication of data in order to improve performances, or introduces several indexing functions that increase the overall size of stored information. Lately, some distributed RDF processing systems have been developed [30, 8, 19, 9]. Trinity.RDF [30] is a distributed GraphDB engine for RDF graph matching: it observes that query processing on RDF graphs (i.e. by SPARQL) requires many graph operations not having locality [23], but relies exclusively on random accesses. Therefore typical disk-based triple-store solutions are not feasible for performing fast random accesses on hard disks. To this aim, among distributed approaches, Trinity.RDF exploits GraphDB technology to store RDF data in a native form and implements a scheduling algorithm to reduce step-by-step the amount of data to analyze during SPARQL query execution. However non-selective queries require many parallel join executions that the generic architecture of Trinity.RDF is not able to integrate, as it is common in MapReduce approaches using Hadoop (e.g., [11]). On the other hand, MapReduce solutions involve a non-negligible overhead, due to the synchronous communication protocols and hob scheduling strategies. Therefore, H2RDF+ [19] builds eight indexes using HBase. It uses Hadoop to perform sort-merge joins during query processing. DREAM [9] 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. In addition, we mention the TriAD distributed system [8], embedding a main-memory architecture, based on the master-slave paradigm. The problem of such system, as in other approaches and as it will be shown experimentally, is that it exhibits complex indexing (i.e., SPO permutation indexing) and

partition schemes (i.e. RDF summary graph), damaging seriously the maintenance of the approach itself and making not possible to exploit completely an in-memory (distributed) engine. Moreover graph data are often maintained in a relational store which is replicated on the disk of each of the underlying nodes of a cluster; managing big attributed graphs on a single machine may be infeasible, especially when the machine's memory is dwarfed by the size of the graph topology.

Contribution. In this paper we propose a novel distributed in-memory approach for SPARQL query processing on *highly unstable* very large datasets. Our objective is to provide a performance-oriented system able to analyze RDF graphs on which *no a priori knowledge* is possible or available, and to avoid collection (exploitation) of complex statistics on initial data and/or frequent past queries. Based on the notion of DOF, the *degree of freedom* of a triple pattern, that is a measure of triple pattern's explicit constraints, we rely on a simple and optimal scheduling algorithm that builds incrementally answers to a SPARQL query. Intuitively, a pattern with no constraints, *i.e.*, constituted only by variables, has the highest DOF, while one constituted by only constants is associated with the lowest DOF. Specifically, our scheduling starts from triple patterns with the lowest degree of freedom, and proceeds in bounding variables incrementally to their values by selecting the triple pattern with the *highest probability of decreasing the search-space*.

As opposed to DBMS-based (single-machine) approaches, our approach avoids any schema or indexing definition over the RDF graph to query, being reindexing impractical for both space and time consumption in a highly volatile environment. Additionally, we highlight as, in a distributed query system, storing RDF data in disk-based triple stores hinders performances, as queries on such graphs are non-local [30], and therefore random access techniques are required to speedup processing. We define a general model of RDF graph based on first principles derived from the *tensor algebra* field. Many real-world data (e.g., knowledge bases, web data, network traffic data, and many others [25, 13, 5]) with multiple attributes are represented as multi-dimensional arrays, called tensors. In analyzing a tensor, tensor decompositions are powerful tools in many data mining applications: correlation analysis on sensor streams [25], latent semantic indexing on DBLP publication data [26], multi-aspect forensics on network data [16], network discovery [5] and so on.

Leveraging such background, this paper proposes a formal tensor representation and endowed with specific operators allowing to perform efficiently our scheduling algorithm for both quick decentralized and centralized massive analysis on large volumes of data—*i.e.*, billions of triples. We strongly rely on an *in-memory distributed approach*, so that our framework may comfortably analyze any given RDF dataset, without any cumbersome processing; in other words, the tensor construction itself is the only processing operation we perform. Our model and operations, inherited by linear algebra and tensor calculus, are therefore theoretically sound, and their implementation exploit the underlying hardware for searching in the solution space. In detail, by applying bit-oriented operations on data, each computational node in our distributed system is able to exploit low-level CPU operations, *e.g.*, 64-bit or 128-bit x86 register opcodes; additionally, operations are carried out in a *cache-oblivious* manner, thus taking advantage of both L1 and L2 caches on modern architectures. Due to the properties of our tensorial model, we are able to dissect tensors (*i.e.*, $\mathcal{R}_i$) representing RDF graphs into several chunks to be processed independently (*i.e.*, by each process p_i), as shown in Figure 1.

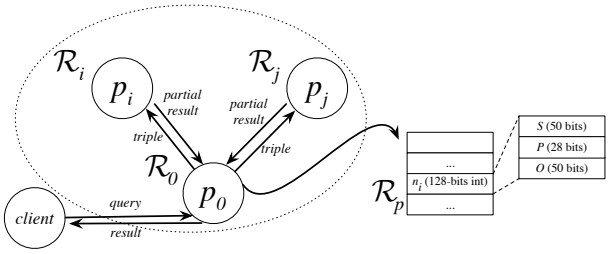


Figure 1: Distributed query processing in TENSORRDF.

Outline. Our manuscript is organized as follows. Section 2 will introduce our general model of a RDF graph, accompanied by a formal tensorial representation, subsequently put into practice in Section 3, where we provide a method for RDF data analysis. Section 4 describes a scheduling algorithm to perform SPARQL queries. Section 5 illustrates the physical modeling of our framework, while Section 6 discusses the complexity of all involved operations. We benchmark our approach with several test beds, and supply the results in Section 7, with the available literature discussed in Section 8. Finally Section 9 sketches some conclusions and future work.

2. RDF AND SPARQL MODELING

This section is devoted to give a rigorous definition of an ontology, with respect to RDF and Semantic Web.

RDF. Data in RDF is built from three disjoints sets $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$ of *IRIs*, *blank nodes*, and *literals*, 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 $s \in \mathcal{I} \cup \mathcal{B}$; $p \in \mathcal{I}$; and $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. RDF is a representation of an ontology. An ontology may be viewed as a set of relations between objects, or more in general, as a *function* that, given two entities s and o, and a relation p, returns a truth value corresponding to the condition of whether or not the two entities are related.

DEFINITION 1 (ONTOLOGY TENSOR). *Let $\mathcal{S}$ be the finite set of* subjects*, $\mathcal{O}$ the finite set of* objects*, and $\mathcal{P}$ the finite set of* predicates*, the* ontology tensor *is a rank-3 tensor $\mathcal{T} : \mathcal{S} \times \mathcal{P} \times \mathcal{O} \longrightarrow \mathbb{B}$, being $\mathbb{B}$ a boolean ring.*

This general definition of ontology tensor must be related to existing representations, in particular with RDF. Hence, we can introduce a direct mapping between the sets $\mathcal{I}$, $\mathcal{B}$, $\mathcal{L}$ and the above definition:

DEFINITION 2 (RDF SETS). *The finite sets $\mathcal{S}$, $\mathcal{P}$, and $\mathcal{O}$ are defined as follows: $\mathcal{S} := \mathcal{I} \cup \mathcal{B}$, $\mathcal{P} := \mathcal{I}$, and $\mathcal{O} := \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$.*

Let us briefly focus on the fact that, by definition, $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$ are finite and *countable*. Therefore, their union is a countable set, and so $\mathcal{S}$, $\mathcal{P}$, and $\mathcal{O}$, consequently. The countability property makes it possible to relate each set to $\mathbb{N}$ via an injective function, *i.e.*, we can "order" all the elements.

DEFINITION 3 (RDF SET INDEXING). *Given the finite countable RDF sets $\mathcal{S}$, $\mathcal{P}$, and $\mathcal{O}$, we introduce their respective* indexing *functions $\mathbb{S}$, $\mathbb{P}$, and $\mathbb{O}$ of* subjects*,* predicates*, and* objects*: $\mathbb{S} : \mathcal{S} \longrightarrow \mathbb{N}$, $\mathbb{P} : \mathcal{P} \longrightarrow \mathbb{N}$, and $\mathbb{O} : \mathcal{O} \longrightarrow \mathbb{N}$.*

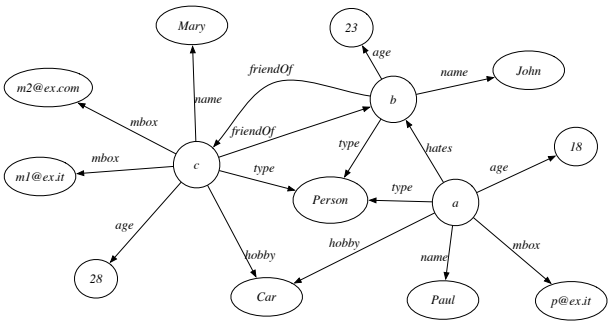


Figure 2: An example of RDF graph G.

The introduced functions, given the finiteness and countability properties are not only injective, but also surjective, *i.e.*, we introduced a bijection between a subset of $\mathbb{N}$, and $\mathcal{S}$, $\mathcal{P}$, $\mathcal{O}$. The *RDF set indexing* functions map elements of RDF sets to a (subset of) natural numbers. Let us focus on the ontology graph G given in Figure 2, constituted of 14 nodes (*i.e.*, 4 resources and 10 literals) and 7 properties. In this case, we have $\mathbb{S}(a) = 1$, $\mathbb{S}(b) = 2$ $\mathbb{S}(c) = 3$, $\mathbb{P}(age) = 1$, $\mathbb{P}(friendOf) = 2$, and so on. Their inverse functions are, being bijections, well defined, *e.g.*, $\mathbb{S}^{-1}(3) = c$.

DEFINITION 4 (RDF TENSOR). *Let G be a RDF graph. The RDF tensor $\mathcal{R}(G) =: \mathcal{R}$ on G is an ontology tensor such that*

$$\mathcal{R} = (r_{ijk}) := \begin{cases} 1, & \langle \mathbb{S}^{-1}(i), \mathbb{P}^{-1}(j), \mathbb{O}^{-1}(k) \rangle \in G\,, \\ 0, & \textit{otherwise}\,. \end{cases}$$

Contrary to adjacency matrices, a tensorial representation allows a simple solution for handling multiple edges between two nodes. Given a RDF tensor, we observe that the majority of its elements will be zero, *i.e.*, the originating graph is loosely connected [1]. It is therefore advisable to employ a *rule notation* to express a tensor, instead of listing all its elements. With the rule notation, we will express a tensor with a list of triples $\{i, j, k\} \to r_{ijk}$, for all $r_{ijk} \neq 0$, and assuming all other elements being zero, if not present in the triples list.

EXAMPLE 1. *Let us consider the RDF graph in Figure 2. The* RDF tensor $\mathcal{R}$ *of G can be therefore given as shown in Figure 3. In the Figure, for typographical simplicity, we omitted*

$$0_{\mathcal{R}} = (0, 0, 0, 0, 0, 0, 0)^t$$

denoted with a dash. A more concise way of expressing the above tensors is by employing the rule notation, i.e.*, assuming zero as the default value, and listing all non-zero elements:*

$$\mathcal{R} = \{\, \{1, 3, 1\} \to 1\,,\ \{1, 4, 3\} \to 1\,,\ \ldots,\ \{3, 1, 13\} \to 1\,\}\,.$$

For instance the element $\{1, 3, 1\} \to 1$ means that there exists in G the triple $\langle \mathbb{S}^{-1}(1), \mathbb{P}^{-1}(3), \mathbb{O}^{-1}(1) \rangle$, i.e.*, $\langle a, hates, b \rangle$.*

SPARQL. Abstractly speaking, a SPARQL query Q mainly is a 5-tuple of the form $\langle qt, RC, DD, G_P, SM \rangle$, where: qt is the *query type*, RC is the *result clause*, DD is the *dataset definition*, G_P is the *graph pattern*, and SM is the *solution modifier*. At the heart of Q there 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 $\mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. In particular the official SPARQL syntax considers operators UNION, OPTIONAL, FILTER and *concatenation* via a point symbol " . " to construct graph patterns.

DEFINITION 5 (GRAPH PATTERN). *A graph pattern G_P is a 4-tuple $\langle \mathbb{T}, f, OPT, U \rangle$ where*

- *$\mathbb{T}$ is a set of* triple patterns *$\{t_1, \ldots, t_n\}$ that may contain a* variable, i.e.*, a symbolic name starting with a ? and can match any node (resource or literal) in the RDF dataset;*
- *f is a* FILTER *constraint using boolean conditions to filter out unwanted query results;*
- *OPT is a set of* OPTIONAL *statements trying to match $\mathbb{T}$, but the whole query does not fail if the optional statements do not match. This set is modeled as G_P;*
- *U is a set of* UNION *statements modeled as G_P.*

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 output of the SPARQL query is then determined by the *query-type*: SELECT, ASK, CONSTRUCT and DESCRIBE.

EXAMPLE 2. *Let us consider the RDF graph shown in Figure 2 and three different SPARQL queries over such graph (*i.e.*, we used simple terms in place of verbose* URI*s).*

```
Q1: SELECT ?x ?y1
    WHERE { ?x type Person. ?x hobby 'CAR'.
            ?x name ?y1. ?x mbox ?y2. ?x age ?z.
             FILTER (xsd:integer(?z) >= 20) }
Q2: SELECT *
    WHERE { {?x name ?y} UNION {?z mbox ?w} }
Q3: SELECT ?z ?y ?w
    WHERE { ?x type Person. ?x friendOf ?y. ?x name ?z.
            OPTIONAL { ?x mbox ?w. } }
```

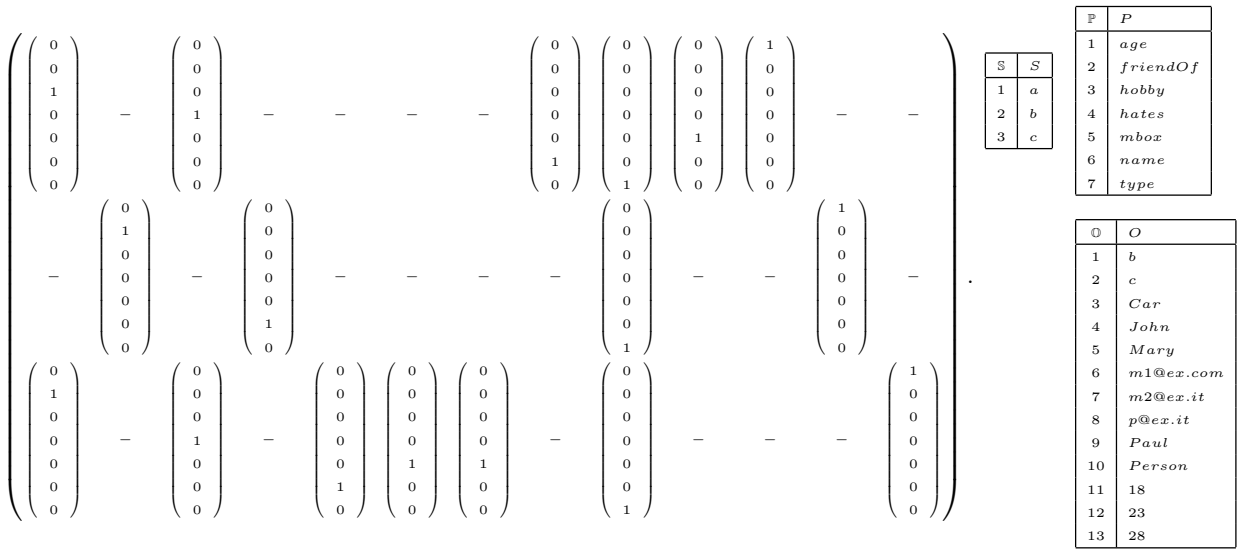
`Q1` *selects* URI *and name of persons having the hobby of cars, a name, a mailbox and an age greater (or equal) than twenty.* `Q2` *selects* URI *and name of persons united to* URI *and mailbox of persons. Finally* `Q3` *selects the name and (in case) the mailbox of all persons having a friend (of which the query returns the* URI *also).*

Referring to Example 2, as illustrated above (in particular refer to definition 5), for instance we model $Q1$ as $\langle$SELECT, $\{?x\}$, $\_$, G_P, $\_\rangle$ with $G_P = \langle \mathbb{T}, f, \_, \_ \rangle$, $\mathbb{T} = \{\langle ?x, type, Person \rangle, \langle ?x, hobby, Car \rangle, \langle ?x, name, ?y1 \rangle, \langle ?x, mbox, ?y2 \rangle, \langle ?x, age, ?z \rangle\}$ and $f = \{?z >= 20\}$.

In [21], the authors analyzed a log of SPARQL queries harvested from the DBPedia SPARQL Endpoint from April to July 2010. The log analysis produced interesting statistics, allowing us to simplify the features of a SPARQL query. In the following we will consider a 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*, employing the operators {AND, FILTER, OPTIONAL, UNION}. This simplification does not compromise the feasibility and generality of the approach.

3. SPARQL DOF MODELING

This section is devoted to the introduction of our approach to SPARQL selection query treatment. In the rest of the paper we will use the term "triple" to indicate also a triple pattern (*i.e.*, a triple can be considered a triple pattern $\langle s, p, o \rangle$, where s, p and o are constants).



$\mathbb{S}$	S
1	a
2	b
3	c

$\mathbb{P}$	P
1	age
2	$friendOf$
3	$hobby$
4	$hates$
5	$mbox$
6	$name$
7	$type$

$\mathbb{O}$	O
1	b
2	c
3	Car
4	$John$
5	$Mary$
6	$m1@ex.com$
7	$m2@ex.it$
8	$p@ex.it$
9	$Paul$
10	$Person$
11	18
12	23
13	28

Figure 3: An example of RDF tensor $\mathcal{R}$ and corresponding RDF Sets Indexing $\mathbb{S}$, $\mathbb{P}$ and $\mathbb{O}$.

3.1 Triple Analysis

A selection query consists of a SELECT clause, followed by a list of *unbounded* variables, *i.e.*, variables whose value is not calculated yet. A WHERE clause states all the conditions the variables must meet, conjunctively.

DEFINITION 6 (DEGREE OF FREEDOM). *The* degree of freedom *of a triple t, $dof : \mathbb{T} \longrightarrow \{+3, +1, -1, -3\}$, is the function defined as: $dof(t) := v-k$, being k and v the number of constants and variables in t, respectively.*

The *degree of freedom*, or DOF, is a measure of a triple's explicit constraints, hence a condition with no constraints (*i.e.*, constituted by variables) has the highest DOF, while one constituted by only constants has the lowest DOF.

EXAMPLE 3. *Referring to Figure 2, the triple $t_1 := \langle a, hates, b\rangle$ is constituted by three constants, and no variables: its DOF is $dof(t_1) = v - k = 0 - 3 = -3$. A triple as $t_2 := \langle a, hates, ?x\rangle$ has two constants, and one variable, namely ?x. Its degree of freedom is $dof(t_2) = v-k = 1-2 = -1$. With $t_3 := \langle ?x, hates, ?y\rangle$ we express a constraint constituted by one constant, $hates$, and two variables. It follows that $dof(t_3) = v - k = 2 - 1 = +1$. Last, let $t_4 := \langle ?x, ?y, ?z\rangle$ be a triple: it is composed by three variables, and no constants. Therefore, $dof(t_4) = v - k = 3 - 0$=+3.*

3.2 Constraint Solving

Due to the fact that triples may have various degrees of freedom, *i.e.*, they may or may not be bound to constants, in the following we shall consider each case, and calculate the results of a triple within our tensorial framework. In particular we employ the vector specification of Kroneker tensor, commonly known as *Kroneker delta* (δ). For instance, given the RDF tensor $\mathcal{R}_{ijk}$, the notation δ_i^2 means that each component in a position different from 2 has value 0, while the component in position 2 has value 1. The number of components in δ_i^2 is equal to the size of the dimension i in $\mathcal{R}_{ijk}$. Referring to the RDF tensor of Figure 3, $\delta_i^2 = (0, 1, 0)$. For the sake of conciseness, we employ a simplified *Einstein's summation convention*: if two adjoined entities share a common index, a summation is implicitly intended, *e.g.*, $\mathcal{R}_{ijk}\, \delta_i^2 := \sum_i \mathcal{R}_{ijk}\, \delta_i^2$.

Degree -3. A triple t with $dof(t) = -3$, is by definition bound to three constants $c_1 \in \mathcal{S}$, $c_2 \in \mathcal{P}$, and $c_3 \in \mathcal{O}$, and therefore the constraint is computed as $\mathcal{R}_{ijk}\, \delta_i^{\mathbb{S}(c_1)}\, \delta_j^{\mathbb{P}(c_2)}\, \delta_k^{\mathbb{O}(c_3)}$.
Degree -1. A triple t with $dof(t) = -1$ possesses two constant values, c_1, and c_2, whose domains are associated with their respective indices i_1 and i_2. The results are hence given by $\mathcal{R}_{ijk}\, \delta_{i_1}^{\mathbb{K}_1(c_1)}\, \delta_{i_2}^{\mathbb{K}_2(c_2)}$ with $\mathbb{K}_1, \mathbb{K}_2 \in \{\mathbb{S}, \mathbb{P}, \mathbb{O}\}$. The result of such computation is a *vector* bound the only variable present in the triple, and, with the rule notation, it may take the form of a list of values.
Degree $+1$. A triple t with $dof(t) = +1$ present two variables, and a single constant c, with the index i_c associated to its domain. So, we may compute the constrains as $\mathcal{R}_{ijk}\, \delta_{i_c}^{\mathbb{K}(c)}$ with $\mathbb{K} \in \{\mathbb{S}, \mathbb{P}, \mathbb{O}\}$. The above equation yields a rank-2 tensor, or in other words, a *matrix*, and promptly interpreted as a list of *couples* when employing the rule notation.
Degree $+3$. A triple t with $dof(t) = +3$ is associated to no constants, and therefore its result is unbounded, *i.e.*, it is computed by returning $\mathcal{R}_{ijk}$.

3.3 Conjunctive Operations

A list of triples in the set $\mathbb{T}$ of a given SELECT query must be satisfied *conjunctively*.

DEFINITION 7 (DISJOINED TRIPLES). *Let $t_1, t_2 \in \mathbb{T}$ be two triples; t_1 and t_2 are* disjoined *if they share no common variables.*

With the above definition, we are able now to focus on the only two cases that may present in a selective SPARQL query.
Disjoined Triples. Given two disjoined triples, their conjunction is simply the union of their bounded variables. By definition, if a variable is bound to an *empty set*, the query yields no results.
Conjoined Triples. Sharing at least one variable, two conjoined constraints t_1 and t_2 produce their results by applying the *Hadamard product* (element-wise multiplication denoted by $\circ$) on the common variables: being $u = (u_i)$ and $v = (v_i)$ we have $z = (z_i) = u \circ v := (u_i \cdot v_i)$.

EXAMPLE 4. *With reference to Figure 2, let t_1 and t_2 be two* conjoined triples*, with $t_1 := \langle ?x, friendOf, c\rangle$, and $t_2 := \langle a, hates, ?x\rangle$. The first triple will be computed as $t_1 := \mathcal{R}_{ijk}\, \delta_j^{\mathbb{P}(friendOf)} \delta_k^{\mathbb{O}(c)}$,*

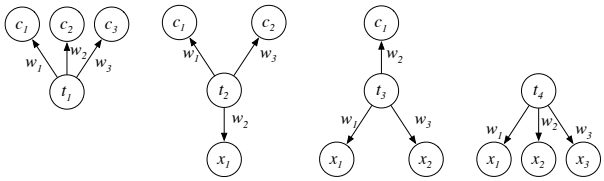


Figure 4: Triples by their *degree of freedom* in an *execution graph*. From left to right, we have DOFs -3, -1, $+1$, and $+3$.

which yields the vector $t_1 := \{\{\mathbb{S}(b)\} \rightarrow 1\}$. *Analogously,* t_2 *will give the result* $t_2 := \mathcal{R}_{ijk}\ \delta_i^{\mathbb{S}(a)}\ \delta_j^{\mathbb{P}(hates)} = \{\{\mathbb{O}(b)\} \rightarrow 1\}$. *Hence,* $t_1 \circ t_2 = \{\{\mathbb{S}(a)\} \rightarrow 1\}$ *is the final result, i.e.* t_1 *and* t_2 *have* b *in common. Conversely, if we have* $t_2 := \langle a, friendOf, ?x\rangle$, *this will yield no results, since* $t_2 := \varnothing$, *and therefore* $t_1 \circ t_2 = \varnothing$.

Anticipating an implementation aspect, we highlight the fact that *conjoined triples* may be computed sequentially for each value in a variable. In other words, if we have a shared variable $?x$ to which there is associated the set of values $\{v_1, v_2\}$, the query shall be processed for v_1, and separately for v_2.

4. QUERY ANSWERING

Given a query, we may now proceed in describing a suitable scheduling algorithm for the analysis of all triples.

4.1 Query Scheduling

Let us first recognize that the final purpose of a scheduling is to determine the *order of execution* of each triple. In turn, this will bind all the variables in the query to their respective values, if any. The degree of freedom of each triple is the primary tool for determining a sequence in which constraints should be solved: it indicates the priority of each triple, with lowest DOFs associated to higher priorities. A *directed acyclic graph* (DAG) may represent an algorithm [3], and in our case, will be employed to visually select all triples for execution.

DEFINITION 8 (EXECUTION GRAPH). *Given a set* $\mathbb{T}$ *of triples, an* execution graph *on* $\mathbb{T}$ *is a weighted directed acyclic graph* $EG = (N, E)$. N *is the set of nodes resulting from* $N_t \cup N_c \cup N_v$, *where* N_t *is the set of triples in* $\mathbb{T}$, N_c *and* N_v *are the sets of* constants *and* variables *associated to the triples of* $\mathbb{T}$, *respectively.* E *is the set of weighted edges connecting triples to their respective constants and variables, with weights representing the domain (*i.e., $\mathcal{S}$, $\mathcal{P}$ *or* $\mathcal{O}$*) of the ending node.*

In order to enhance readability, we present the execution graph in a three-layered fashion, as depicted in Figure 4. The center layer contains all triples N_t, the top layer the constants N_c, and the bottom layer the variables N_v.

EXAMPLE 5. *With reference to the query* Q1 *of Example 2 and Figure 4, we rewrite* Q1 *as follows. The triple* $t_1 := \langle ?x, type, Person\rangle$ *has DOF* -1 *and is represented by the second graph of Figure 4, with* $c_1 = type$, $c_2 = Person$, *and their respective weights are* $w_1 = \mathcal{P}$, $w_2 = \mathcal{S}$, *and* $w_3 = \mathcal{O}$. *Hence, the computation of* t_1 *is given by* $\mathcal{R}_{ijk}\ \delta_j^{\mathbb{P}(type)}\ \delta_k^{\mathbb{O}(Person)}$, *giving birth to a vector of elements, bound to the variable* $?x$. *The triple* $t_2 := \langle ?x, hobby, car\rangle$ *has DOF* -1 *and is represented similarly to* t_1. *Therefore we may compute* t_2 *as* $\mathcal{R}_{ijk}\ \delta_j^{\mathbb{P}(hobby)}\ \delta_k^{\mathbb{O}(car)}$. *The triples* $t_3 := \langle ?x, name, ?y1\rangle$, $t_4 := \langle ?x, mbox, ?y2\rangle$, $t_5 := \langle ?x, age, ?z\rangle$ *have DOF* $+1$ *and are represented by the third graph of Figure 4. For instance, referring to* t_3, *we have only one constant* $c_1 = name$ *with weight* $w_2 = \mathcal{P}$,

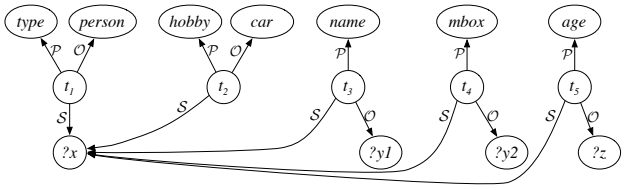


Figure 5: An example of execution graph.

and two variables $x_1 = ?x$ *and* $x_2 = ?y1$; *their respective weights are* $w_1 = \mathcal{S}$, $w_3 = \mathcal{O}$. *Hence,* t_3 *is computed as* $\mathcal{R}_{ijk}\ \delta_j^{\mathbb{P}(name)}$: *the matrix resulting from the computation is the set of couples associated to the variables* $?x$ *and* $?y1$. *Figure 5 shows the final execution graph built from* Q1.

The scheduling for the execution of a SELECT SPARQL query is therefore dynamically determined. In our framework, given a set $\mathbb{T}$ of triples, the scheduling algorithm can be sketched as follows:

1. Determine the DOF of each triple $t_i \in \mathbb{T}$;
2. Select the triple $\widetilde{t} \in \mathbb{T}$ with lowest DOF;
3. Execute $\widetilde{t}$ as described in Section 3.2;
4. Bind all variables in the triples of $\mathbb{T}$ conjunctively;
5. Remove $\widetilde{t}$ from the list;
6. If $\mathbb{T} = \varnothing$ *stop*; else proceed to *step* 1.

In the previous scheduling schema, we may encounter triples with the same DOF. In this case, in Step 2 we shall select the triple which raises the DOF of the largest numer of triples in a query, excluding itself. Suppose for instance that our triple patterns are as follows: *?x name ?y*, *?x hobby ?u*, *?u color ?z*, and finally the pattern *?u model ?w*. In the example we may notice as every triple has DOF equal to $+1$. However, analyzing the prospect DOFs, we notice as the first will promote only the second query through *?u*, both the third and fourth will affect two patterns—the second and fourth—while the second will affect all queries, and hence it is selected for processing.

EXAMPLE 6. *This example will elucidate the process of query answering via DOF analysis. With reference to the RDF tensor* $\mathcal{R}$ *described in Figure 3, let the SPARQL query under scrutiny be Q1 = ?x type Person. ?x hobby 'CAR'. ?x name ?y1. ?x mbox ?y2. ?x age ?z.; as discussed above, we have five triples* t_1, t_2, t_3, t_4, *and* t_5. *Given our five constraints we hence proceed in analyzing their* degrees of freedom, *resulting in* $dof(t_1) = dof(t_2) = 1$, *and* $dof(t_3) = dof(t_4) = dof(t_5) = +1$. *We may proceed now in executing the query. First, let us remind that any variable is currently* unbounded, *or in other words, unassociated to any value. The first triple to be computed is between* $t_1 := \langle ?x, type, Person\rangle$ *and* $t_2 := \langle ?x, hobby, car\rangle$, *having the lowest value of DOF, equal to* -1. *In this case we start from* t_1. *Therefore we determine the values associated to this constraint. This triple produces a* vector *of values to be bound to* $?x$:

$$t_1 := \mathcal{R}_{ijk}\ \delta_j^{\mathbb{P}(type)}\ \delta_k^{\mathbb{O}(person)} = \\ = \{\{\mathbb{S}(a)\} \rightarrow 1, \{\mathbb{S}(b)\} \rightarrow 1, \{\mathbb{S}(c)\} \rightarrow 1\}\ .$$

This computation returns the set X *of values* $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ *to be associated to the variable* $?x$. *Therefore we have to reanalyze the*

degree of freedom of the remaining triples. In this case we have $dof(t_2) = -3$ *and* $dof(t_3) = dof(t_4) = dof(t_5) = -1$, i.e.*, the variable* $?x$ *is promoted to the role of constant. The next triple to be computed is* t_2*, having the highest value of DOF, equal to* -3*. Therefore, for each* $x_z \in X$*, the triple yields the boolean value*

$$t_2 := \mathcal{R}_{ijk}\ \delta_i^{\mathbb{S}(x_z)}\ \delta_j^{\mathbb{P}(hobby)}\ \delta_k^{\mathbb{O}(car)} = 1\,.$$

Since the outcome of the computation is true *for* $x_z \in \{a, c\}$*, the set* X *is filtered accordingly, and the query processing may proceed. Proceeding our scheduling,* t_3 *has to be processed similarly to* t_1*. This triple produces a* vector *of values to be bound to* $?y1$*:*

$$\begin{aligned} t_3 \ &:= \mathcal{R}_{ijk}\ \delta_i^{\mathbb{S}(a)}\ \delta_j^{\mathbb{P}(name)} \cup \mathcal{R}_{ijk}\ \delta_i^{\mathbb{S}(c)}\ \delta_j^{\mathbb{P}(name)} = \\ &= \{\,\{\mathbb{O}(Paul)\} \to 1, \{\mathbb{O}(Mary)\} \to 1\,\}\,. \end{aligned}$$

Owing to the non-emptiness of the previous computation, we proceed to the last triples t_4 *and then* t_5 *computed as* t_3 *producing the vectors of values to be bound to* $?y2$ *and* $?z$*, respectively.*

$$\begin{aligned} t_4 := \quad & \{\,\{\mathbb{O}(p@ex.it)\} \to 1, \{\mathbb{O}(m1@ex.it)\} \to 1, \\ & \{\mathbb{O}(m2@ex.com)\} \to 1\,\}\,. \\ t_5 := \quad & \{\,\{\mathbb{O}(18)\} \to 1, \{\mathbb{O}(28)\} \to 1\,\}\,. \end{aligned}$$

In both the triples all values in the set X*,* i.e.*, a and c, provide non-empty results in the computation. Finally we apply the filter to the values associated to the variable* $?z$*,* i.e.*,* $?z \geq 20$*. Consequently, we have to filter* t_5 *to* $\{\{\mathbb{O}(28)\} \to 1\}$ *and then the set* X*,* i.e.*,* $X = \{c\}$*. Since all triples were processed, the scheduling stops. Moreover we bind the set* $Y1$ *of values associated to* $?y1$ *to* X *obtaining* $\{Mary\}$*. Because the result clause of Q1 is* $?x\ ?y1$*, we return the so-generated* X *and* $Y1$*.*

In the rest of this section we provide an implementation of our scheduling algorithm to perform both "conjunctive" and "non-conjunctive" SELECT SPARQL queries.

4.2 Conjunctive Pattern with Filters

The so-called conjunctive pattern with filters (CPF) uses only the operators AND and FILTER. Therefore our scheduling algorithm takes as input a set $\mathbb{T}$ of triple patterns $t_1, \ldots, t_m$, a filter f, a set X_v of result clause variables $?x_1, \ldots, ?x_n$ and a RDF tensor $\mathcal{R}$, in terms of sum of p chunks $\mathcal{R}_i$; p is the number of processes on p hosts, while $\mathcal{R}_i$ is the slice of $\mathcal{R}$ corresponding to the set of triples in the i-th host. The output is a set $\mathcal{X}_I$ of instances $X_1, \ldots, X_n$, that is, each X_i contains values to be associated to the variable $?x_i$ such that all constraints $t_1, \ldots, t_m$ are satisfied. The set $\mathcal{X}_I$ is computed as shown in Algorithm 1.

More in detail, we initialize a map `V` where the keys are all the variables occurring in the triples of $\mathbb{T}$ while to each key we associate a set of values, i.e. at the beginning an empty set (lines [1-2]). The procedure `getVariables` is responsible to extract all variables from the triple patterns in $\mathbb{T}$. For instance referring to the query `Q1` of Example 2 on the RDF graph of Figure 2, we have $\mathbb{T} = \{t_1, t_2, t_3, t_4, t_5\}$ as described above, $f = ?z \geq 20$, $X_v = \{?x, ?y1\}$ and the RDF tensor $\mathcal{R}$ illustrated in Figure 3. The map `V` is initialized to $\{\langle ?x, \varnothing\rangle, \langle ?y1, \varnothing\rangle, \langle ?y2, \varnothing\rangle, \langle ?z, \varnothing\rangle\}$.

We organize $\mathbb{T}$ as a priority queue (i.e. high priority corresponds to low DOF associated to a constraint t). Then we extract a constraint t from $\mathbb{T}$ until $\mathbb{T}$ is not empty and the computation of t produces a non-empty result (lines [4-12]). In particular, a broadcast mechanism sends t and V to all hosts, which compute t on the own $\mathcal{R}_\rangle$. The resulting values associated to the variables occurring in t are included in the set `V` and then filtered by applying f (*i.e.*, the procedure `Filter` is responsible of such task) in terms of a map operation: being $u = (u_i)$ and f a suitable function, $v = (v_i) = \mathrm{map}(f, u) := (\,f(u_i)\,)$. Consequently, we update the DOFs of each triple in $\mathbb{T}$ (*i.e.*, the procedure `Update` is responsible of such task). At the end, if all triples brought result we return the sets of values associated to the variables in X_v, otherwise we return an empty set.

Algorithm 1: Execution of a SPARQL query

```
Input  : T = {t_1, ..., t_m}, f, X_v = {?x_1, ..., ?x_n},
         R = R_1 + ... + R_p
Output: X_I = {X_1, ..., X_n}
1  V ← ∅;
2  foreach ?x ∈ getVariables(T) do V.put(?x,∅);
3  proceed ← true;
4  while (T is not empty) ∧ (proceed) do
5      t ← T.dequeue();
6      broadcast(t);
7      proceed ← reduce(Application(t, V, R_i), OR);
8      if (proceed) then
9          Update(T, V);
10         Filter(V,f);
11         foreach ?x ∈ getVariables({t}) do
12             reduce(V.get(?x), sum);
13 if (proceed) then
14     foreach ?x ∈ X_v do X_I ← X_I ∪ V.get(?x);
15 else X_I ← ∅;
16 return X_I;
```

Algorithm 2: Tensor application of a triple

```
Input  : t, V = {⟨?x_1, X_1⟩, ..., ⟨?x_z, X_z⟩}, R
Output: boolean
1  if isVariable(t.s) then S ← V.get(t.s);
2  else S ← S ∪ {t.s};
3  if isVariable(t.p) then P ← V.get(t.p);
4  else P ← P ∪ {t.p};
5  if isVariable(t.o) then O ← V.get(t.o);
6  else O ← O ∪ {t.o};
7  switch dof(t,V) do
8      case -3
9          return CASETHREE(t, S, P, O, V, R);
10     case -1
11         return CASEONE(t, S, P, O, V, R);
12     case +1
13         return CASEMINUSONE(t, S, P, O, V, R);
14     case +3
15         V.put(t.s, getValues(R_ijk 1̄_j 1̄_k));
16         V.put(t.p, getValues(R_ijk 1̄_i 1̄_k));
17         V.put(t.o, getValues(R_ijk 1̄_i 1̄_j));
18         return true;
19     otherwise
20         return false;
```

The computation of a triple t is performed by the procedure `Application`. Since we are in a distributed environment, we exploit a `reduce` function that takes as input a set of values and an operator to combine such values. Since `Application` returns a boolean value (*i.e.* true if the tensor application was able to compute t), the `reduce` function takes all boolean values from each host and combine them through the `OR` logic operator (line [7]). Similarly, if the result of such `reduce` is true, then we combine all the values retrieved for each variable $?x$ in t by the hosts by reducing them though a `sum` operator, *i.e.* union, (lines [11-12]).As shown in Algorithm 2, it takes as input the triple t, the map `V` and

the tensor slice $\mathcal{R}$. In this procedure we have to evaluate the DOF of t: how many constants (or variables to which there exists a non-empty set associated in V) and how many variables (to which an empty set is associated in V). We use the notation $t.s$, $t.p$ and $t.o$ to access to subject, property and object of a triple t, respectively. The procedure `isVariable` evaluates if a component of t is a variable: we extract all values associated in the previous computations and we put them in the sets S, P and O. Otherwise S, P and O contain the constants $t.s$, $t.p$ and $t.o$, respectively. W.r.t the DOF, we call the corresponding procedure implementing the tensor application on $\mathcal{R}$.

Case -3. The triple t has DOF -3; in this case we have to filter all $s \in S$, $p \in P$ and $o \in O$ such that there does not exist a triple $\langle s, p, o \rangle$ in $\mathcal{R}$. As illustrated in Algorithm 3, we iterate on S, P and O to compute the set D of elements to filter. If all S, P and O are not empty we can proceed with our scheduling.

Algorithm 3: Case with DOF -3

```
Input  : t, S, P, O, V, R
Output: boolean
D ← ∅;
foreach s ∈ S do
    foreach p ∈ P do
        foreach o ∈ O do
            if R_{S(s)P(p)O(o)} = 0 then
                D ← D ∪ {s};
                D ← D ∪ {p};
                D ← D ∪ {o};
            else
                D ← D − {s};
                D ← D − {p};
                D ← D − {o};
S ← S − D;
P ← P − D;
O ← O − D;
if isVariable(t.s) then V.put(t.s,S);
if isVariable(t.p) then V.put(t.p,P);
if isVariable(t.o) then V.put(t.o,O);
return (S ≠ ∅ ∧ P ≠ ∅ ∧ O ≠ ∅);
```

Case -1. The triple t has DOF -1; in this case t provides only one variable. As shown in Algorithm 4, the procedure `roleVariable` evaluates which component of t is a variable (*i.e.*, `'s'` for subject, `'p'` for property, `'o'` for object). We have to compute the application $\mathcal{R}_{ijk}\ \delta_{i_1}^{\mathbb{K}_1(c_1)}\ \delta_{i_2}^{\mathbb{K}_2(c_2)}$ on the two constants c_1 and c_2 of t. For instance if `roleVariable`(t) is `'s'` then we employ the constants $p \in P$ and $o \in O$ (*i.e.*, coming from the previous computations) and we compute $\mathcal{R}_{ijk}\ \delta_j^{\mathbb{P}(p)}\ \delta_k^{\mathbb{O}(o)}$. The procedure `getValues` retrieves the values associated to the resulting vector and put them in the set X. Finally we associate X to the variable $t.s$ in V. If the resulting set X is not empty we can proceed with our scheduling.

Case $+1$. The triple t has DOF $+1$; in this case t provides only one constant. As shown in Algorithm 5, the procedure `roleConstant` evaluates which component of t is a constant.

We have to compute $\mathcal{R}_{ijk}\ \delta_{i_c}^{\mathbb{K}(c)}$. For instance if `roleConstant`(t) is `'s'`, we have to extract all predicates and objects associated to the elements $e \in S$ (coming from the previous computations in case). Therefore we compute $\mathcal{R}_{\mathbb{S}(e)jk}\,\bar{1}_k$ and $\mathcal{R}_{\mathbb{S}(e)jk}\,\bar{1}_j$; $\mathcal{R}_{\mathbb{S}(e)jk}$ returns a matrix fixing the dimension i to $\mathbb{S}(e)$ while $\bar{1}_j$ ($\bar{1}_k$) is a vector with all components 1 and with length equal to the size of

Algorithm 4: Case with DOF -1

```
Input  : t, S, P, O, V, R
Output: boolean
X ← ∅;
switch roleVariable(t) do
    case 's'
        foreach p ∈ P do
            foreach o ∈ O do
                X ← X ∪ getValues(R_{ijk} δ_j^{P(p)} δ_k^{O(o)});
        V.put(t.s,X);
    case 'p'
        foreach s ∈ S do
            foreach o ∈ O do
                X ← X ∪ getValues(R_{ijk} δ_i^{S(s)} δ_k^{O(o)});
        V.put(t.p,X);
    case 'o'
        foreach s ∈ S do
            foreach p ∈ P do
                X ← X ∪ getValues(R_{ijk} δ_i^{S(s)} δ_j^{P(p)});
        V.put(t.o,X);
    otherwise
        return false;
return X ≠ ∅;
```

the dimension j (k). All the properties (E_1) and objects (E_2) are then associated to $t.p$ and $t.o$ in the map V. If E_1 and E_2 are non empty our scheduling can proceed.

Case $+3$. If the triple t has DOF $+3$ we have to extract all subjects ($\mathcal{R}_{ijk}\bar{1}_j\bar{1}_k$), properties ($\mathcal{R}_{ijk}\bar{1}_i\bar{1}_k$) and objects ($\mathcal{R}_{ijk}\bar{1}_i\bar{1}_j$) from $\mathcal{R}$.

4.3 Non-Conjunctive Pattern with Filters

The so called non conjunctive pattern with filters (non-CPF) employs OPTIONAL and UNION, beyond AND and FILTER. In this case our scheduling algorithm has to perform disjoined triples.

Union. Given a query $\langle RC, G_P \rangle$ with $G_P = \langle \mathbb{T}, f, \_, U \rangle$ (*i.e.*, $U = \langle \mathbb{T}_U, f_U, \_, \_ \rangle$), we perform our scheduling algorithm on the triples of both $\mathbb{T}$ and $\mathbb{T}_U$, separately. Finally we make the union of all $\mathcal{X}_I$. For instance let us consider the query Q2 of Example 2. We have $\mathbb{T} = \{t_1\}$ and $\mathbb{T}_U = \{t_2\}$, where $t_1 := \langle ?x, name, ?y \rangle$ and $t_2 := \langle ?z, mbox, ?w \rangle$. From $\mathbb{T}$ we generate $\mathcal{X}_I = \{\{a, b, c\}, \{Paul, John, Mary\}\}$ while from $\mathbb{T}_U$ we have $\mathcal{X}_I = \{\{a, c\}, \{p@ex.it, m1@ex.it, m2@ex.com\}\}$.

The final result is $\mathcal{X}_I = \{\ \{a, b, c\},\ \{Paul, John, Mary\},$
$\{\ p@ex.it,\ m1@ex.it,$
$m2@ex.com\}\ \}$.

Optional. Given a query $\langle RC, G_P \rangle$ with $G_P = \langle \mathbb{T}, f, OPT, \_ \rangle$ (*i.e.*, $OPT = \langle \mathbb{T}_{OPT}, f_{OPT}, \_, \_ \rangle$), we perform our scheduling algorithm on the triples of both $\mathbb{T}$ and $\mathbb{T} \cup \mathbb{T}_{OPT}$, separately. Finally we make the union of all $\mathcal{X}_I$. For instance let us consider the query Q3 of Example 2, we have $\mathbb{T} = \{t_1, t_2, t_3\}$ and $\mathbb{T}_{OPT} = \{t_4\}$, where $t_1 := \langle ?x, type, Person \rangle$, $t_2 := \langle ?x, friendOf, ?y \rangle$, $t_3 := \langle ?x, name, ?z \rangle$ and $t_4 := \langle ?x, mbox, ?w \rangle$. From $\mathbb{T}$ we generate $\mathcal{X}_I = \{John, Mary\}, \{b, c\}\}$, while from $\mathbb{T} \cup \mathbb{T}_{OPT}$ we have $\mathcal{X}_I = \{\{Mary\}, \{b\}, \{m1@ex.it,\ m2@ex.com\}\}$. The final result is $\mathcal{X}_I = \{\{John, Mary\}, \{b, c\}, \{m1@ex.it,\ m2@ex.com\}\}$.

Of course, both the graph patterns U and OPT can be more complex, i.e. with other UNION or OPTIONAL statements; in this

Algorithm 5: Case with DOF +1

Input : $t, S, P, O, \mathsf{V}, \mathcal{R}$
Output: boolean

$E_1 \leftarrow \varnothing$;
$E_2 \leftarrow \varnothing$;
switch `roleConstant` (t) **do**
  case 's'
    foreach $e \in S$ **do**
      $E_1 \leftarrow E_1 \cup \texttt{getValues}(\mathcal{R}_{\mathbb{S}(e)jk}\bar{1}_k)$;
      $E_2 \leftarrow E_2 \cup \texttt{getValues}(\mathcal{R}_{\mathbb{S}(e)jk}\bar{1}_j)$;
    V.put $(t.p, E_1)$;
    V.put $(t.o, E_2)$;
  case 'p'
    foreach $e \in P$ **do**
      $E_1 \leftarrow E_1 \cup \texttt{getValues}(\mathcal{R}_{i\,\mathbb{P}(e)k}\bar{1}_k)$;
      $E_2 \leftarrow E_2 \cup \texttt{getValues}(\mathcal{R}_{i\,\mathbb{P}(e)k}\bar{1}_i)$;
    V.put $(t.s, E_1)$;
    V.put $(t.o, E_2)$;
  case 'o'
    foreach $e \in O$ **do**
      $E_1 \leftarrow E_1 \cup \texttt{getValues}(\mathcal{R}_{ij\,\mathbb{O}(e)}\bar{1}_j)$;
      $E_2 \leftarrow E_2 \cup \texttt{getValues}(\mathcal{R}_{ij\,\mathbb{O}(e)}\bar{1}_i)$;
    V.put $(t.s, E_1)$;
    V.put $(t.p, E_2)$;
  otherwise
    return false;
return $(E_1 \neq \varnothing \wedge E_2 \neq \varnothing)$;

case we apply the above procedure recursively. Concluding, once our scheduling algorithm produced $\mathcal{X}_I$, we demand to a *front-end* task the presentation of results in terms of tuples, conforming to the result clause of the query.

5. IMPLEMENTATION

Our prime objective is to provide a general storage, in memory data structures and operators for distributed query processing in our framework. As detailed in Section 7, we make use of a clustering file system, *i.e.*, the Lustre [15] file system. In our system we were not allowed low-level administration, and therefore we could not tune Lustre for optimal performance by imposing *ad-hoc* parameters: we therefore chose a file format that could exploit a parallel distributed access on a shared file system, in particular, we chose the *Hierarchical Data Format* version 5, or HDF5. The HDF5 data format [10] is a binary storage that allows a hierarchical organization of large datasets. It supports platform-independent binary data types, multidimensional arrays, and grouping in order to provide more articulated data structures. In comparison to standard DBMSs, recent developments in the analysis of biological dataset highlighted that databases are effective in dealing with string-based data, whereas management is more difficult for complex numerical structures (cf. Millard et al., Nature [17]). Limits of HDF5 on the top of a Lustre file system is beyond present-day realistic constraints: the maximum archive size is imposed by HDF5, 10^{18} bytes, or 1000 PB.

Permanent Storage. Several data structures have been proposed for (sparse) tensors, *i.e.*, multidimensional arrays [24]. A common representation for sparse matrices is the *Compressed-Row Storage* format, or briefly CRS, with its dual CCS—*Compressed-Column Storage* (cf. [6]). Such matrices with nnz non-zero entries, are represented by means of three different arrays: one of length nnz representing all stored entries, in row-major order for CRS (column-major for CCS); one array of length equal to the number of rows,

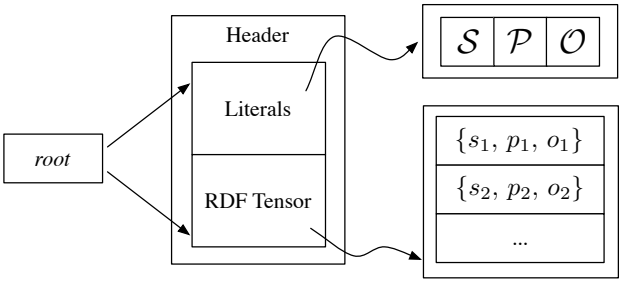


Figure 6: Data storage within the HDF5 file format.

containing the indexes of the first element of each row (or column); finally, the column index vector of each non-zero element. Literature describes numerous data structures related to CRS, aimed at tensor representation: in essence, elements are stored by *sorting* indexes, and subsequently memorizing index vectors as CRSs, a technique commonly known as *slicing*. It is clear as the *order of sorting* is crucial: being $\mathcal{R}_{ijk}$ a tensor sorted on the i-th coordinate, calculating $\mathcal{R}_{ijk}v_i$ is optimized, but $\mathcal{R}_{ijk}v_k$ is not [2]. Moreover, all CRS descendants suffer from the same drawbacks of their ancestor: they highly depend on the assumption that elements are *evenly distributed among rows*. Moreover, such data structures are *bounded to particular dimensions* of a tensor, or in simpler terms, changing the size of a coordinate—*e.g.*, introducing a new property—is a burdensome operation (cf. [14]).

We therefore chose another common data format to model RDF graphs as tensors, the *Coordinate Sparse Tensors* [2], or CST. This format, already introduced in Section 2, memorizes tensors as a list of tuples: we memorize a list of nnz entries, describing the entry value and coordinates, parallel to the description illustrated in Figure 3. The main advantage of this organization is its simplicity and adaptability: it is *order independent* with respect to the RDF tuples, allows fast parallel access to data, requires no particular index sorting on coordinates, and allows run-time dimension changes with the addition of new entries.

As previously said, we chose HDF5 as the hierarchical permanent storage medium. The *root* of the HDF5 storage will contain a header with pointers to two main data structured: the *Literals* list and the *RDF tensor* itself. The former, as exemplified in Figure 6, contains the list of all literals needed by a user to identify objects in the RDF graph: in other words, it incorporates the list of literals and constants found in RDF groups $\mathcal{S}$, $\mathcal{P}$, and $\mathcal{O}$, hence, it implicitly defines $\mathbb{S}$, $\mathbb{P}$, and $\mathbb{O}$. The latter group is the RDF tensor, stored as a list of triples by means of Coordinate Sparse Tensor representation. By definition if a triple is not present in the list, then its associated boolean value is *false*, hence omitted.

Parallel Operations. Our storage exploits the performance boost obtainable by a binary interface and numerical data. We are able, therefore, to *split* data over different processes, so that I/O overhead may be further ameliorated. The CST data structure does not rely on any particular ordering, and therefore given p processes, and being n the number of triples stored in the RDF tensor, we may simply distribute evenly n/p 3-tuples on each process, owing to the associative and distributive properties of linear forms. In fact, given $\mathcal{R}_{ijk}v_\ell$, with $\ell \in \{i, j, k\}$, it is perfectly licit to obtain the result as

$$\mathcal{R}_{ijk}v_\ell = \left(\sum_{z=1}^{p} R^z_{ijk}\right) v_\ell = \sum_{z=1}^{p} \left(R^z_{ijk}\, v_\ell\right) , \qquad (1)$$

```
typedef __uint128_t rdft;

// Returns an 128-bit integer from components
inline rdft toStorage(long s, long p, long o)
{
    return (static_cast<rdft>(s) << 0x4E) |
           (static_cast<rdft>(p) << 0x32) |
           (static_cast<rdft>(o));
}

auto it = std::find_if(p.begin(), p.end(),
    [](rdft d){ return d &
                toStorage(42, 0xFFFFFFF, 256); });
```

Figure 7: A representative search with 128-bits integer encoding, searching for a triple matching $\langle \mathbb{S}^{-1}(42), ?x, \mathbb{O}^{-1}(256) \rangle$.

being $\mathcal{R}^z_{ijk}$ the z-th tensor partition, *i.e.*, a set of n/p triples assigned to each process. As the reader may perceive, a parallel implementation of all the operators introduced in the previous sections is straightforward, being inherently data-parallel [3].

We remind that our environment shall handle *highly unstable data-sets*, and as such we do not perform any indexing, as said before. Therefore, each and every computational node in our environment will read a portion of all RDF triples independently of any order, *i.e.*, as they appear in the dataset. Having access to the Lustre distributed file system, each node in the cluster may read its contiguous portion of data, *i.e.*, z-th processor will read n/p triples, with offset equal to zn/p, with $z \in \mathbb{N} \leq p$ being the processor unique id, and p being equal to the number of available processes. Hence, each process will hold a fragment of the whole tensor $\mathcal{R}$, being the part in itself a valid sparse tensor.

Equation (1) shows that the application of a tensor to a vector may be conducted independently on each process, multiplying the partial tensor $\mathcal{R}^z$ with the vector v. In order to reconstruct the complete result $\mathcal{R}\,v$, we shall sum all the contribution computed by each process, an operation that, in distributed terms, is named *reduction*. The reduction operator combines all data from every process with an *associative operation*, in our case, we employ reductions over boolean rings and vector spaces (cf. Algorithm 1), and are carried on communicating among processes using binary trees [22].

Tensor Application. Our implementation has the primary objective of exploiting the underlying hardware for accelerating tensor applications. This paragraph briefly describes the implementation of tensor application described in Section 3.2, and leveraged on each computational node as reported in the previous paragraph.

The tensorial framework we developed relies on the latest ISO C++11 specification, utilizing an unordered vector as the main computational node in-memory data structure. The vector contains all triples stored in a single computational node, encoded as a single *128-bit unsigned integer*; each integer is decomposed bit-by-bit, *i.e.*, interpreted as a sequence of bits representing, in order, $\mathbb{S}$, $\mathbb{P}$, and $\mathbb{O}$. In our implementation, we reserved 50 bits for subject and object, and 28 bits for the property, as detailed in the function `toStorage` in Figure 7. Applying the tensor to a vector falls into one of the four cases exposed in Section 3.2, dependently on each triple pattern's degree of freedom: each DOF case shall multiply the tensor with one or more Dirac deltas (*i.e.*, it is a generalization of Kronecker delta). However, we may conduct those operations simultaneously by scanning the vector for matching triples, encoded in a single 128-bit integer.

Scanning the vector, which is guaranteed to be allocated in a single contiguous block of memory, leverages memory caches and minimizes *cache misses*. In other words, the naïve data structure allows us to employ a simple bit-wise *cache-oblivious* search algorithm [7]. In order to optimize queries, searching is performed by utilizing the bit-wise *and* operator: a SPARQL triple pattern is encoded as a single 128-bit integer, shifting their numerical values and *or*-ing them; free variables, as for instance in Figure 7, are represented by a sequence of bit set to 1. Our implementation optimises further comparisons by exploiting CPU-level SSE2/SSE3 instructions, available on every modern processors, and used as accelerators in several computational fields, as for instance bioinformatics or databases. Hence, a triple $\langle s, p, ?x \rangle$ is encoded combining $\mathbb{S}^{-1}(s)$, $\mathbb{P}^{-1}(p)$, and a sequence of 50 bit set in a single 128-bit integer number, treated via XMM 128-bit capable registers.

6. THEORETICAL ANALYSIS

The ensuing paragraphs analyze the theoretical complexity of all the operations involved in SPARQL queries, according to Sections 3 and 4. In the following, we will employ the notation $nnz(M)$, with M being the rank-3 RDF tensor under analysis, denoting the number of its non-zero values—analogously $nnz(v)$ yields the number of non-zero entries of a vector v.

Insertion. The assembly of a sparse tensor requires the basic operation of inserting an element into the list of non-zero values, if not present. The operation has therefore a complexity of $O\left(nnz(M)\right)$, and $O\left(nnz(v)\right)$ for vectors.

Deletion and Update. Such basic actions mimic the above insertion operation, and therefore have an asymptotic complexity of $O\left(nnz(M)\right)$.

Hadamard Product. The Hadamard product of two vectors $u \circ v$ has a complexity of $O\left(nnz(u)\ nnz(v)\right)$.

Tensor Application. For a suitable vector v, the tensor application on the ℓ-th dimention $M_{ijk}v_\ell$, with $\ell \in \{i, j, k\}$, has asymptotic complexity of $O\left(nnz(M)\right)$, as detailed in [2].

Mapping. Mapping a function on a tensor or a vector, *i.e.*, filtering information, has clearly a linear complexity $O(nnz(M))$ and $O(nnz(v))$. All other non-zero element will be mapped once, and eventually inserted in the result, if the mapping yields a *non-false* value.

Scheduling. The naïve scheduler described in Section 4.1 is optimal. First, let us consider a (computational) *cost function* of a triple pattern. In our environment, where no statistical information about the SPARQL dataset is available, we may assume the degree of freedom of each triplet as an indicator of the supposed computational cost. Let $s^* = \{s_1, \ldots, s_t\}$ be the optimal scheduled triple patterns in a SPARQL query, *i.e.*, the scheduling with the minimal cost; if the actual scheduling s differs from s^* therefore at least one step i, the algorithm chose a different triple pattern, *i.e.*, $s_i \neq s^*_i$, with $dof(s_i) > dof(s^*_i)$. However, this contradicts the step 2 of the algorithm presented in Section 4.1: if this were the case, the chosen scheduled triplet would have been s^*_i, hence, we have that necessarily $s^* = s$.

7. RESULTS

We implemented our framework into TENSORRDF, a C++ system using OpenMPI v1.8 library for answering SPARQL queries over RDF datasets. We performed a series of experiments aimed at

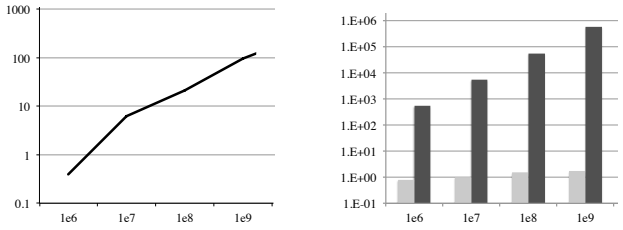


Figure 8: Data loading times in seconds (a) and query memory footprint expressed in MB (b). Light gray bars refer to system memory overhead, while dark gray ones show data set size.

evaluating the performance of our approach, with the main results detailed in this section.

Benchmark Environment. We deployed TENSORRDF on a cluster, wherein each machine is supported by 48 GB DDR3 RAM, 16 CPUs 2.67 GHz Intel Xeon (*i.e.*, each with 4 cores and 8 threads), running Scientific Linux 5.7, with the TORQUE Resource Manager process scheduler. The system is provided with the Lustre file system v2.1, coupled with HDF5 library v1.8.7. The performance of our systems has been measured with respect to data loading, memory footprint, and query execution time with reference to z processes running on z hosts. We evaluated the performance of TENSORRDF comparing with centralized triple stores Sesame, Jena-TDB, and BigOWLIM, and open-source systems BitMat [1] and RDF-3X [18], as well as distributed MapReduce-RDF-3X [11], Trinity.RDF [30] and TriAD-SG [8] (i.e., since neither Trinity.RDF and TriAD are openly available, we will refer to the running times reported in [30] and [8]).

In our experiments, we employed the popular LUBM synthetic benchmark (i.e. LUBM-4450 which consists of about 800M triples) and two real-life datasets: DBPEDIA v3.6, *i.e.*, 200M triples loaded into the official SPARQL endpoint, and BTC-12, the dataset of 2012 Billion Triples Challenge, that is more than 1000M triples. For each dataset we involved a set of test queries. For DBPEDIA we wrote 25 queries of increasing complexity (available at https://www.dropbox.com/sh/pz0i67s9ohbpb9t/oEGo-J8yui). Such queries involve SELECT SPARQL queries embedding *concatenation* " . ", FILTER, OPTIONAL and UNION operators. We use such dataset for comparison with centralized approaches. In this case, we set up TENSORRDF on a single machine. Referring to BTC-12, we exploit the test queries defined in [18]; in both LUBM-4450 and BTC-12 we have SELECT SPARQL queries involving only *concatenation*. This two last datasets are used for comparison with distributed approaches.

Loading and Memory Footprint. Referring to data management, we are able to achieve three main goals. First, we are able to perform loading *without any particular relational schema*, when compared to triple store approaches, where a schema coupled with appropriate indexes have to be maintained by the system. Second, owing to the flexibility of CST we are capable of *modifying substantially the tensor dimension*, *i.e.*, introducing novel literals in either RDF sets is a trivial operation: whereas a DBMS must perform a re-indexing, we may carry this operation without any additional overhead. As last objective, owing to the distributivity and associativity properties of our theoretical model, we are able to distribute data and computational power over different hosts, allowing also parallel access to the data. In this case, we refer to a 12-server cluster deployment. Data loading times are 45, 110 and 130 seconds for DBPEDIA, LUBM-4450, and BTC-12, respectively. In particular, as showed in Figure 8(a), data loading times are 0.395, 6.194, 21.068, and 129.699 seconds, for all examined dimensions in BTC-12. Another significant advantage of our system relies in memory consumption. In particular, always referring to a 12-server cluster deployment, the overall memory overhead needed to maintain a distributed tensor representation of RDF data almost constant, and amounts to circa 1 MB of RAM. Referring to BTC-12, the total distributed memory consumption for our tests were 549.3 MB, 5.391, 44.121, and 332.918 GB for all examined dimensions; both memory overhead and RAM occupation are depicted in Figure 8(b). On average, all triple store systems require a data space 10 times greater, BitMat 5 times greater, RDF-3X, Trinity.RDF and TriAD-SG 2-3 times greater.

Query Analysis. We ran the queries ten times and measured the average response time (including the I/O times) in ms. On disk-based systems, we performed both *cold-cache* and *warm-cache* experiments. Figure 9 illustrates the response times on DBPEDIA in a 1-server cluster (*i.e.*, centralized environment). On average, Sesame and Jena-TDB perform poorly, BigOWLIM and BitMat better, and RDF-3X is competitive. TENSORRDF outperforms all competitors, in particular RDF-3X by a large margin. TENSORRDF is 18 times better than RDF-3X, 128 times on the maximum (*i.e.*, Q21). In particular the queries involving OPTIONAL and UNION operators (*e.g.*, Q20) require the most complex computation: triple stores, *i.e.*, BigOWLIM, Sesame and Jena-TDB, depend on the physical organization of indexes, not always matching the joins between patterns. RDF-3X provides a permutation of all combinations 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. On the other hand, we exploit the sparsity of our tensor and compute in parallel map functions, tensor applications, and Hadamard products. 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 10 shows the memory usage (KB) to query DBPEDIA in a 1-server cluster. On the average, all queries (also the most complex) require very few bytes of memory (*i.e.*, dozens of KBytes), whereas all competitors require dozens of MB.

As distributed systems, we measured times for TENSORRDF, TriAD-SG (i.e., TriAD using Summary Graph), Trinity.RDF and MapReduce-RDF-3X (simply MR-RDF-3X) on a 12-server cluster with 1GBit LAN connection for both LUBM-4450 and BTC-12. We highlight that we used a set of SELECT queries embedding only *concatenation*, on which competitors exploit own physical indexing in a profitable way. Figure 11 presents the results, showing that our system performs 9 times better than MR-RDF-3X and 5 times better thanTrinity.RDF for LUBM-4450, while 100 times better than MR-RDF-3X and 1,5 times better thanTrinity.RDF for BTC-12. TriAD-SG is the most competitive system: however for queries non selective (i.e. LUBM-4450) our system is comparable to TriAD-SG, while for selective queries (i.e. BTC-12) our system outperforms TriAD-SG. This is due by exploiting the algebraic properties of our tensorial representation that allows us to process in parallel triples in small chunks and by embedding DOF evaluation to speed-up selective triples execution. Referring to memory consumption, querying LUBM-4450 and BTC-12 presents a behavior (*i.e.*, dozens of KBytes) similar to DBPEDIA, while competitors dozens of MB. For space constraints, we do not report the diagram.

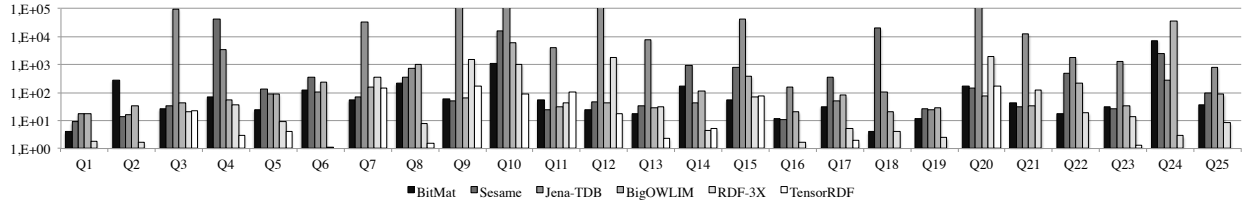


Figure 9: Times on DBpedia in ms: different gray scale bars refer each system, *i.e.*, from BitMat (black), to TENSORRDF (white). Response time of TENSORRDF for Q1, Q2, Q6, Q16, Q18, Q19, Q21, Q22 and Q25 was less than 1 ms.

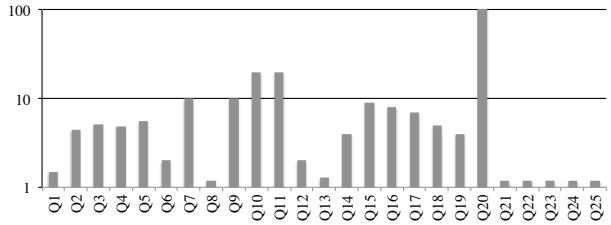


Figure 10: Memory Usage to query DBPEDIA in KB.

We also performed *warm-cache* experiments, but we are unable to report them due to space constraints; however while TENSORRDF improves performance from milliseconds to microseconds (*e.g.*, from 1 ms to 0.1 μs), the other competitors improves performance in milliseconds magnitude (*e.g.*, from 100 ms to 1 ms). As last experiment, we tested TENSORRDF's scalability, reported in Figure 12 with a limited representative number of queries (*i.e.*, Q3, Q6 and Q7 in BTC-12 since they are the most complex). As the dimension of our problem increases, from 500MB to 300GB, the time increases from approximately 10^{-3} ms, to 10^{1} ms for the largest dimension, *i.e.*, for a number of triples of 10^{9}.

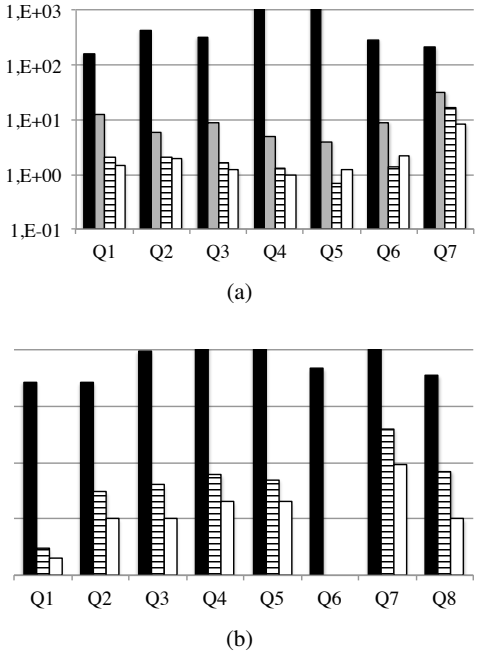


Figure 11: Response Times in ms on LUBM-4450 (a), and BTC-12 (b), for MR-RDF-3X (in black), Trinity.RDF (in gray), TriAD-SG (striped filled) and TENSORRDF (in white).

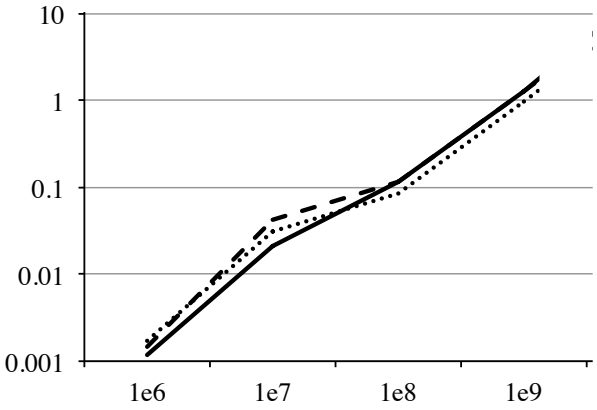


Figure 12: Scalability on BTC-12: times (ms) are plotted against number of triples. Solid, dotted and dashed lines refer respectively to Q4, Q7, and Q8.

8. RELATED WORKS

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: developing native storage systems with ad-hoc optimizations, and making use of traditional DBMSs (such as relational and object-oriented). Native storage systems (such as OWLIM, or RDF-3X [18]) are more efficient in terms of load and update time, whereas the adoption of mature data management systems exploit consolidate and effective optimizations. Indeed, native approaches need re-thinking query optimization and transaction processing techniques. However, the number of required self-joins makes this approach impractical, and the optimizations introduced to overcome this problem have proven to be query-dependent, or to introduce significant computational overhead (cf. [4]).

On the querying side, current research in SPARQL pattern processing (cf. [18, 8] and [27]) 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. Nevertheless, a keen observation of SPARQL queries from real-life logs [12]showed that *non-conjunctive* queries are employed in non-negligible numbers, providing detailed statistics. An interesting approach was given in [1] by Atre et al., which start from a dense (*i.e.*, not sparse) tensorial representation, and generate all possible combinations of two dimensional matrices of relations, named BitMats, discarding some pairings such as Object-Property "since based on [our] experience, usage of those BitMats is rare"; finally, they *compress* row-wise with a RLE scheme, amounting on a total of $2|\mathcal{P}| + |\mathcal{S}| + |\mathcal{O}|$ matrices. Query computation require

significant workload and post-processing.

On the other hand, our approach exploits algebraic properties for both storage and computation, and owing to the DOF sorting, is able to schedule triplets in an efficient order; additionally, our framework may deal with non-conjunctive queries and data change. As illustrated in Section 7, all the above approaches are optimized for centralized analysis requiring significant amount of resources, both in terms of memory and storage. In opposite, Trinity.RDF [30] and TriAD [8] exploit in-memory frameworks to provide efficient general-purpose query processing on RDF in a distributed environment. However, as shown in Section 7, the efficiency on such proposals is strictly depending on the logical and physical organization of data, and on the complexity of the 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, allowing RDF dataset to change comfortably and to distribute computation over several hosts, without any *a priori* knowledge about RDF dataset or querying statistics.

9. 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 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 Cloud environments.

10. REFERENCES

[1] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *WWW*, pages 41–50, 2010.

[2] B. W. Bader and T. G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[3] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[4] S. M. 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.

[5] I. N. Davidson, S. Gilpin, O. T. Carmichael, and P. B. Walker. Network discovery via constrained tensor analysis of fmri data. In *KDD*, pages 194–202, 2013.

[6] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[7] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297, 1999.

[8] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *SIGMOD*, pages 289–300, 2014.

[9] M. Hammoud, D. A. Rabbou, R. Nouri, S. Beheshti, and S. Sakr. DREAM: distributed RDF engine with adaptive query planner and minimal communication. *PVLDB*, 8(6):654–665, 2015.

[10] G. Heber. HDF5 meets, challenges, and complements the DBMS. In *XLDB*, 2011.

[11] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.

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

[13] T. G. Kolda and J. Sun. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*, pages 363–372, 2008.

[14] C.-Y. Lin, Y.-C. Chung, and J.-S. Liu. Efficient data compression methods for multidimensional sparse array operations based on the ekmr scheme. *IEEE Trans. Comput.*, 52:1640–1646, 2003.

[15] M. W. Margo, P. A. Kovatch, P. Andrews, and B. Banister. An analysis of state-of-the-art parallel file systems for linux. In *5th International Conference on Linux Clusters: The HPC Revolution 2004*, 2004.

[16] K. Maruhashi, F. Guo, and C. Faloutsos. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *ASONAM*, pages 203–210, 2011.

[17] B. L. Millard, M. Niepel, M. P. Menden, J. L. Muhlich, and P. K. Sorger. Adaptive informatics for multifactorial and high-content biological data. *Nature Methods*, 8(6):487–492, 2011.

[18] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD*, pages 627–640, 2009.

[19] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H_2rdf+: an efficient data management system for big RDF graphs. In *SIGMOD*, pages 909–912, 2014.

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

[21] F. Picalausa and S. Vansummeren. What are real sparql queries like? In *SWIM - SIGMOD Workshop*, page 7, 2011.

[22] R. Rabenseifner. Optimization of collective reduction operations. In *Computational Science-ICCS 2004*, pages 1–9. Springer, 2004.

[23] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *ICDT*, pages 4–33, 2010.

[24] M. P. Sears, B. W. Bader, and T. G. Kolda. Parallel implementation of tensor decompositions for large data analysis. In *SIAM*, 2009.

[25] J. Sun, S. Papadimitriou, and P. S. Yu. Window-based tensor analysis on high-dimensional and multi-aspect streams. In *ICDM*, pages 1076–1080, 2006.

[26] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *KDD*, pages 374–383, 2006.

[27] M.-E. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently joining group patterns in sparql queries. In *ESWC*, pages 228–242, 2010.

[28] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[29] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: a fast and compact system for large scale rdf data. *PVLDB*, 6(7):517–528, 2013.

[30] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *PVLDB*, pages 265–276, 2013.

[31] S. Zhang, S. Li, and J. Yang. Gaddi: distance index based subgraph matching in biological networks. In *EDBT*, pages 192–203, 2009.