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 systems;

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, 11]) 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 hbd 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, 51]) 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 social model, we are able to dissect tensors (i.e., RDF summary graph), damaging seriously the maintenance of the approach itself and making not possible to relate each set to a boolean ring.

**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 $I$, $B$, and $L$ of IRI, blank nodes, and literals, respectively. All information in RDF is represented by triples of the form $(s, p, o)$, where $s$ is called the subject, $p$ is called the predicate, and $o$ is called the object. To be valid, it is required that $s \in I \cup B$; $p \in I$; and $o \in I \cup B \cup L$. 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 $S$ be the finite set of subjects, $O$ the finite set of objects, and $P$ the finite set of predicates, the ontology tensor is a rank-3 tensor $T : S \times P \times O \rightarrow B$, being $B$ 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 $I$, $B$, $L$ and the above definition:

**Definition 2 (RDF Sets).** The finite sets $S$, $P$, and $O$ are defined as follows: $S := I \cup B$, $P := I$, and $O := I \cup B \cup L$.

Let us briefly focus on the fact that, by definition, $I$, $B$, and $L$ are finite and countable. Therefore, their union is a countable set, and so $S$, $P$, and $O$, 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 $S$, $P$, and $O$, we introduce their respective indexing functions $S$, $P$, and $O$ of subjects, predicates, and objects: $S : S \rightarrow N$, $P : P \rightarrow N$, and $O : O \rightarrow N$. 

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
</tr>
<tr>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
</tr>
<tr>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

**Figure 1:** Distributed query processing in TENSORRDF.
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 \( S, P, \mathcal{O} \). 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 \( \mathcal{S}(a) = 1, \mathcal{S}(b) = 2, \mathcal{S}(c) = 3, \mathcal{P}(\text{age}) = 1, \mathcal{P}(\text{friendOf}) = 2 \), and so on. Their inverse functions are, being bijections, well defined, e.g., \( S^{-1}(1) = a \).

**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

\[
\mathcal{R} \equiv (r_{ijk}) := \begin{cases} 1, & (S^{-1}(i),P^{-1}(j),O^{-1}(k)) \in G, \\ 0, & \text{otherwise}. \end{cases}
\]

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 \( \{i, j, k\} \rightarrow r_{ijk} \), 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

\[
0 \equiv (0, 0, 0, 0, 0, 0)'
\]

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:

\[
\mathcal{R} = \{ \{1, 3, 1\} \rightarrow 1, \{1, 4, 3\} \rightarrow 1, \ldots, \{3, 1, 13\} \rightarrow 1 \}.
\]

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

**SPARQL.** Abstractly speaking, a SPARQL query \( Q \) mainly is a 5-tuple of the form \( (q_t, RC, DD, GP, SM) \), where: \( q_t \) is the query type, \( RC \) is the result clause, \( DD \) is the dataset definition, \( GP \) is the graph pattern, and \( SM \) is the solution modifier. At the heart of \( Q \) there lies the graph pattern \( GP \) 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 \( T \cup B \cup L \). In particular the official SPARQL syntax considers operators \( \text{UNION}, \text{OPTIONAL}, \text{FILTER} \) and \( \text{concatenation} \) via a point symbol “.” to construct graph patterns.

**Definition 5 (Graph Pattern).** A graph pattern \( GP \) is a 4-tuple \( (\mathcal{T}, f, OPT, U) \) where

- \( \mathcal{T} \) 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 \( \mathcal{T} \), but the whole query does not fail if the optional statements do not match. This set is modeled as \( GP \);
- \( U \) is a set of UNION statements modeled as \( GP \).

The result clause identifies which information to return from the query. It returns a table of variables (occurring in \( GP \)) 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 URIs).

\[
Q1: \text{SELECT ?y1} \\
\text{WHERE} \{ ?x \text{type Person. ?x hobby "CAR".} \}
\]

\[
Q2: \text{SELECT *} \\
\text{WHERE} \{ (?x \text{ name ?y1.} ?x \text{ mbox ?y2. ?x \text{ age ?z.} \} \}
\]

\[
Q3: \text{SELECT ?y ?w} \\
\text{WHERE} \{ (?x \text{ name ?y.} ?x \text{ friendOf ?y.} ?x \text{ name ?z.} \} \}
\]

**Example 3.** Referring to Example 2, as illustrated above (in particular refer to definition 5), for instance we model \( Q1 \) as \( (\text{SELECT, } \{ ?x, ?y1 \}, \{ GP_1, \ldots, GP_n \}) \) with \( GP = \{ (\mathcal{T}, f, OPT, U) \} \), i.e.,

\[
\begin{align*}
& \mathcal{T} = \{ \{ ?x \text{ name ?y1.} \} \} \\
& \mathcal{U} = \{ \{ ?x \text{ name ?y1.} \} \}
\end{align*}
\]

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 \( (RC, GP) \), 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 DOP 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).
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 $t$, $\text{dof}(t) : \mathbb{T} \rightarrow \{+3, -1\}$, is the function defined as: $\text{dof}(t) = v - k$, 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 $t_1 := \langle a, \text{hates}, b \rangle$ is constituted by three constants, and no variables: its DOF is $\text{dof}(t_1) = v - k = 0 - 3 = -3$. A triple as $t_2 := \langle a, \text{hates}, ?x \rangle$ has two constants, and one variable, namely $?x$. Its degree of freedom is $\text{dof}(t_2) = v - k = 1 - 2 = -1$. With $t_3 := \langle ?x, \text{hates}, ?y \rangle$ we express a constraint constituted by one constant, hates, and two variables. It follows that $\text{dof}(t_3) = v - k = 2 - 1 = +1$. Last, let $t_4 := \langle ?x, ?y, ?z \rangle$ be a triple: it is composed by three variables, and no constants. Therefore, $\text{dof}(t_4) = v - k = 3 - 0 = +3$.

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 ($\delta$). For instance, given the RDF tensor $R_{ijk}$, the notation $\delta_i^j$ 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 $\delta_i^j$ is equal to the size of the dimension $i$ in $R_{ijk}$.

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., $R_{ijk} \delta_i^j := \sum_k R_{ijk} \delta_i^k$.

**Degree $+3$.** A triple $t$ with $\text{dof}(t) = +3$, is by definition bound to three constants $c_1 \in S$, $c_2 \in P$, and $c_3 \in O$, and therefore the constraint is computed as $R_{ijk} \delta_i^j \delta_j^k$.

**Degree $+1$.** A triple $t$ with $\text{dof}(t) = +1$ poses 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 $R_{ijk} \delta_i^{(c_1)} \delta_j^{(c_2)}$ with $R_{ijk}$, $K \in \{S, P, O\}$. 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 $+3$.** A triple $t$ with $\text{dof}(t) = +3$ is associated to no constants, and therefore its result is unbounded, i.e., it is computed by returning $R_{ijk}$.

3.3 Conjointive Operations

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

**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 $\odot$) on the common variables: being $u = (u_i)$ and $v = (v_i)$ we have $z = (z_i) = u \odot v := (u_i \cdot v_i)$.

**Example 4.** With reference to Figure 2, let $t_1$ and $t_2$ be two conjoined triples, with $t_1 := \langle ?x, \text{friendOf}, c \rangle$, and $t_2 := \langle a, \text{hates}, ?x \rangle$. The first triple will be computed as $t_1 := R_{ijk} \delta_i^{(\text{friendOf})} \delta_j^{(c)}$,
which yields the vector \( t_1 := \{ \{ S(b) \} \to 1 \} \). Analogously, \( t_2 \) will give the result \( t_2 := R_{ijk} \delta^i_{(\text{hats})} \delta^j_{(\text{bats})} = \{ \{ O(b) \} \to 1 \} \).

Hence, \( t_1 \circ t_2 = \{ \{ S(a) \} \to 1 \} \) is the final result, i.e. \( t_1 \) and \( t_2 \) have \( b \) in common. Conversely, if we have \( t_2 := \{ a, \text{friendOf}, ?x \} \), this will yield no results, since \( t_2 := \emptyset \), and therefore \( t_1 \circ t_2 = \emptyset \).

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 \(?s\) 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 scheduling algorithm [3], and in our case, will be employed to visually select all triples for execution.

**Definition 8 (Execution Graph).** Given a set \( T \) of triples, an execution graph on \( \mathbb{T} \) is a weighted directed acyclic graph \( EGC = (N, E) \). \( N \) is the set of nodes resulting from \( N_1 \cup N_2 \cup N_3 \), where \( N_1 \) is the set of triples in \( T \), \( N_2 \) are the sets of constants and variables associated to the triples of \( T \), respectively. \( E \) is the set of weighted edges connecting triples to their respective constants and variables, with weights representing the domain (i.e., \( S \), \( P \), or \( O \)) 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 \( N_1 \), the top layer the constants \( N_2 \), and the bottom layer the variables \( N_3 \).

**Example 5.** With reference to the query \( Q_1 \) of Example 2 and Figure 4, we rewrite \( Q_1 \) as follows. The triple \( t_1 := \{ ?x, \text{type}, \text{Person} \} \) has DOF \(-1\) and is represented by the second graph of Figure 4, with \( c_1 = \text{type}, c_2 = \text{Person} \), and their respective weights are \( w_1 = P, w_2 = S, w_3 = O \). Hence, the computation of \( t_1 \) is given by \( R_{ijk} \delta^i_{(\text{type})} \delta^j_{(\text{Person})} \). The first layer of this graph is the set \( N_1 \) of triples, and the second layer is the set of constants and variables associated to \( N_1 \). The third layer is the set of weighted edges connecting \( N_1 \) to \( N_2 \) and \( N_3 \), respectively. The weights are determined by the values associated to the variables \(?x\), \(?y\), \(?z\), \(?a\), \(?b\), \(?c\), \(?d\), \(?w\), \(?z\).

The scheduling for the execution of a SELECT SPARQL query is therefore dynamically determined. In our framework, given a set \( T \) 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 \( t_i \in \mathbb{T} \) with lowest DOF; execute it as described in Section 3.2.
3. Execute \( t_i \) as described in Section 3.2.
4. Bind all variables in the triples of \( \mathbb{T} \) conjunctively;
5. Remove \( t_i \) from the list;
6. If \( \mathbb{T} = \emptyset \) 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 number of triples in a query, excluding itself. Suppose for instance that our triple patterns are as follows: \(?x, \text{name}, ?y\), \(?x, \text{hobby}, ?a\), \(?a, \text{color}, ?c\), and finally the pattern \(?a, \text{model}, ?w\). 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 \(?a\), 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 \( R \) described in Figure 3, let the SPARQL query under scrutiny be \( Q_1 = ?x \text{ type Person}. ?x \text{ hobby 'CAR}'. ?x \text{ name '?y1'. ?x mash ?y2'. ?x age '?z2'; } \) 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 \( \text{doF}(t_1) = \text{doF}(t_2) = 1, \) and \( \text{doF}(t_3) = \text{doF}(t_4) = \text{doF}(t_5) = +1 \). We may proceed now in executing the query. First, let us remind that any variable is currently unbound, or in other words, unassociated to any value. The first triple to be computed is between \( t_1 := \{ ?x, \text{type, Person} \} \) and \( t_2 := \{ ?x, \text{hobby, car} \} \), 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 \(?x\):

\[
\begin{align*}
t_1 := R_{ijk} & \delta^i_{(\text{type})} \delta^j_{(\text{Person})} \\
& = \{ \{ S(a) \} \to 1, \{ S(b) \} \to 1, \{ S(c) \} \to 1 \} .
\end{align*}
\]

This computation returns the set \( X \) of values \( \{ a, b, c \} \) to be associated to the variable \(?x\). For instance, we have to reanalyze the
degree of freedom of the remaining triples. In this case we have
dof(t_2) = −3 and dof(t_4) = dof(t_5) = −1, i.e., the
variable ?z 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_2 ∈ X, the triple yields the boolean value
t_2 := R_{ijk} δ^{(o)}_{j} δ^{(bobby)}_{j} δ^{(car)}_{i} = 1.

Since the outcome of the computation is true for x_2 ∈ {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_2.
This triple produces the vectors of values to be bound to ?y1:
t_3 := R_{ijk} δ^{(o)}_{j} δ^{(name)}_{j} =
\{ \{ \Omega(Paul) \} \rightarrow 1, \{ \Omega(Mary) \} \rightarrow 1 \}.

Owing to the non-emptiness of the previous computation, we pro-
cceed 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.
t_4 := \{ \{ \Omega(p@ex.it) \} \rightarrow 1, \{ \Omega(m2@ex.it) \} \rightarrow 1,
\{ \Omega(m2@ex.com) \} \rightarrow 1 \}.
t_5 := \{ \{ \Omega(18) \} \rightarrow 1, \{ \Omega(28) \} \rightarrow 1 \}.

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 ≥ 20. Conse-
quently, we have to filter t_3 to \{ \{ \Omega(28) \} \rightarrow 1 \} and then the set
X, i.e., X = \{c\}. Since all triples were processed, the scheduling
stops. Moreover we bind the set Y 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-conjunc-
tive” 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 T of triple patterns t_1, . . . , t_m, a filter f, a set
X_r of result clause variables ?x_1, . . . , ?x_n and a RDF tensor R,
in terms of sum of p chunks R_i, p is number of processes on p
hosts, while R_i is the slice of R corresponding to the set of triples
in the i-th host. The output is a set \{X_i\} of instances X_1, . . . , X_n,
that is, each X_i contains values associated to the variable
?x_i that all constraints t_1, . . . , t_m are satisfied. The set \{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 T while to each key we asso-
ciate 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 T. For instance referring
to the query Q1 of Example 2 on the RDF graph of Figure 2, we
have T = \{t_1, t_2, t_3, t_4, t_5\} as described above, f = t_2 ≥ 20,
X_r = \{?x, ?y, ?z\} and the RDF tensor R illustrated in Figure 3.
The map V is initialized to \{(?x, ?z), (?y, ?z), (?y, ?z), (?z, ?z)\}.

We organize T as a priority queue (i.e. high priority corresponds
to low DOF associated to a constraint t). Then we extract a con-
straint t from T until T is not empty and the computation of t pro-
duces a non-empty result (lines [4–12]). In particular, a broadcast
mechanism sends t and V to all hosts, which compute t on the

---

**Algorithm 1: Execution of a SPARQL query**

| Input: Τ = \{t_1, . . . , t_m\}, \text{f}, X_r = \{?x_1, . . . , ?x_n\}. |
| Output: \{X_1, . . . , X_n\}. |

1. V ← ∅.
2. foreach \(t_r \in \text{getVariables}(T)\) do V.put(?r,∅);
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 \(?x \in \text{getVariables}([f])\) do
12. reduce(V.get(?x), sum);
13. if (proceed) then
14. \(\text{foreach } ?x ∈ X_r \text{ do } X_r ← X_r ∪ V.get(?x)\);
15. else \(X_r ← ∅\);
16. return \(X_r\).

own \(R_i\). The resulting values associated to the variables occur-
ing 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,
\(v = (v_i) = \text{map}(f, u) := (f(u_i))\). Consequently, we update the
DOFs of each triple in \(T\) (i.e., the procedure Update is respon-
sible of such task). At the end, if all triples brought result we return
the sets of values associated to the variables in \(X_r\), otherwise we
return an empty set.

**Algorithm 2: Tensor application of a triple**

| Input : t, V = \{(?x_1, X_1), . . . , (?x_n, X_n)\}, \text{R} |
| Output: boolean |

1. if IsVariable(?t) then return CASEMINUSONE(t, S, P, O, V, R);
2. else \(S ← S ∪ \{t\}\);
3. if IsVariable(?p) then \(P ← V.get(?p)\);
4. else \(P ← P ∪ \{t\}\);
5. if IsVariable(?t) then \(O ← V.get(?o)\);
6. else \(O ← O ∪ \{t\}\);
7. Switch-do(t, V) do
8. [case -3 return CASETHREE(t, S, P, O, V, R);]
9. [case -1 return CASEONE(t, S, P, O, V, R);]
10. [case +1 return CASEMINUSONE(t, S, P, O, V, R);]
11. [case +3 V.put(?t, V.getValues(\{R_i\}1^{1}_{1}));]
12. [V.put(?p, V.getValues(\{R_i\}1^{1}_{1}));]
13. [V.put(?o, V.getValues(\{R_i\}1^{1}_{1}));]
14. return true;
15. otherwise 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 com-
pute \(t\)), 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 re-
ducing them through 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

239
the tensor slice $R$. 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 $R$.

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 $R$. 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 $-3$

<table>
<thead>
<tr>
<th>Input</th>
<th>$t.s$, $t.p$, $t.o$, $V$, $R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Boolean</td>
</tr>
<tr>
<td>$D \leftarrow \emptyset$</td>
<td></td>
</tr>
<tr>
<td>foreach $s \in S$ do</td>
<td></td>
</tr>
<tr>
<td>foreach $p \in P$ do</td>
<td></td>
</tr>
<tr>
<td>foreach $o \in O$ do</td>
<td></td>
</tr>
<tr>
<td>if $\mathcal{R}_{t.s}(p,o) = 0$ then</td>
<td></td>
</tr>
<tr>
<td>$D \leftarrow D \cup {s}$</td>
<td></td>
</tr>
<tr>
<td>$D \leftarrow D \cup {p}$</td>
<td></td>
</tr>
<tr>
<td>$D \leftarrow D \cup {o}$</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>$D \leftarrow D \setminus {s}$</td>
<td></td>
</tr>
<tr>
<td>$D \leftarrow D \setminus {p}$</td>
<td></td>
</tr>
<tr>
<td>$D \leftarrow D \setminus {o}$</td>
<td></td>
</tr>
<tr>
<td>$S \leftarrow S \setminus D$</td>
<td></td>
</tr>
<tr>
<td>$P \leftarrow P \setminus D$</td>
<td></td>
</tr>
<tr>
<td>$O \leftarrow O \setminus D$</td>
<td></td>
</tr>
<tr>
<td>if isVariable($t.s$) then $V.put(t.s, S)$</td>
<td></td>
</tr>
<tr>
<td>if isVariable($t.p$) then $V.put(t.p, P)$</td>
<td></td>
</tr>
<tr>
<td>if isVariable($t.o$) then $V.put(t.o, O)$</td>
<td></td>
</tr>
<tr>
<td>return $(S \neq \emptyset \land P \neq \emptyset \land O \neq \emptyset)$</td>
<td></td>
</tr>
</tbody>
</table>

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 empty our scheduling can proceed.

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

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 $\langle RC, GP \rangle$ with $GP = \langle \top, f \cup OP T, \cup \rangle$ (i.e., $U = \langle U, \cup, \cup, \cup \rangle$), we perform our scheduling algorithm on the triples of both $\top$ and $U$, separately. Finally we make the union of all $X_j$. For instance let us consider the query $\forall q$ of Example 2. We have $T = \{t_1\}$ and $T_U = \{t_2\}$, where $t_1 := \langle ?x, name, ?y \rangle$ and $t_2 := \langle ?z, mbox, ?w \rangle$. From $T$ we generate $X_T = \{\{a, b, c\}, \{Paul, John, Mary\}\}$ while from $T_U$ we have $X_U = \{\{a, b\}, \{Paul, John, Mary\}, \{p@ex.it, m1@ex.it, m2@ex.com\}\}$. The final result is $X_r = \{\{a, b, c\}, \{Paul, John, Mary\}, \{p@ex.it, m1@ex.it, m2@ex.com\}\}$.

Optional. Given a query $\langle RC, GP \rangle$ with $GP = \langle \top, f, OPT, \cup \rangle$ (i.e., $OPT = \langle OPT, fOPT, \cup \rangle$), we perform our scheduling algorithm on the triples of both $\top$ and $OPT$, separately. Finally we make the union of all $X_j$. For instance let us consider the query $\forall q$ of Example 2, we have $T = \{t_1, t_2, t_3\}$ and $T_{OPT} = \{t_4\}$, where $t_1 := \langle ?x, type, Person \rangle$, $t_2 := \langle ?x, friendOf, ?y \rangle$, $t_3 := \langle ?x, name, ?z \rangle$ and $t_4 := \langle ?z, mbox, ?u \rangle$. From $T$ we generate $X_T = \{\{John, Mary\}, \{b\}, \{m1@ex.it, m2@ex.com\}\}$. The final result is $X_r = \{\{John, Mary\}, \{b\}, \{m1@ex.it, m2@ex.com\}\}$.

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, V, R
Output: boolean

1. E₁ ← ∅; 1
2. E₂ ← ∅; 2
3. switch roleConstant(t) do
   4. case 's' do
      5. foreach e ∈ S do
         6. E₁ ← E₁ ∪ getValues(\(R_{e(i,j)k}\)₁;
         7. E₂ ← E₂ ∪ getValues(\(R_{e(i,j)k}\)₂;
         8. V.put(t.p, E₁);
         9. V.put(t.s, E₂);
      10. case 'p' do
         11. foreach e ∈ P do
            12. E₁ ← E₁ ∪ getValues(\(R_{p(i,j)k}\)₁;
            13. E₂ ← E₂ ∪ getValues(\(R_{p(i,j)k}\)₂;
            14. V.put(t.p, E₁);
            15. V.put(t.s, E₂);
      16. case 'o' do
         17. foreach e ∈ O do
            18. E₁ ← E₁ ∪ getValues(\(R_{o(i,j)k}\)₁;
            19. E₂ ← E₂ ∪ getValues(\(R_{o(i,j)k}\)₂;
            20. V.put(t.s, E₁);
            21. V.put(t.o, E₂);
      22. otherwise do
         23. return false;
   24. return (E₁ ≠ ∅ ∧ E₂ ≠ ∅);

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, 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 slicing is crucial: being \(R_{ijk}\) a tensor sorted on the i-th coordinate, calculating \(R_{ijk}v_ℓ\) is optimized, but \(R_{ijk}v_\ell\) is not [22]. 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 \(S, \mathcal{P}\), and \(\mathcal{O}\), hence, it implicitly defines \(\mathcal{S}, \mathcal{P}\), and \(\mathcal{O}\). 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 tuples stored in the RDF tensor, we may simply distribute evenly \(\frac{n}{p}\) 3-tuples on each process, owing to the associative and distributive properties of linear forms. In fact, given \(R_{ijk}v_\ell\), with \(\ell \in \{i, j, k\}\), it is perfectly licit to obtain the result as

\[
R_{ijk}v_\ell = \left( \sum_{z=1}^{p} R_{ijk}^z \right) v_\ell = \sum_{z=1}^{p} \left( R_{ijk}^z v_\ell \right),
\]

Figure 6: Data storage within the HDF5 file format.
typedef __uint128_t rdft;
// Returns an 128-bit integer from components
inline rdft toStorage(long s, long p, long ...)

OpenMPI v1.8 library for answering SPARQL queries over RDF datasets. We performed a series of experiments aimed at...

...and every computational node in our en...

...each node in the cluster may read its...

...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 ∈ N ≤ 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 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 R_{i,j} with the vector v. In order to reconstruct the complete result 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, ⊗, ∨, and ⊕. 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 optimizes 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 (s, p, ?x) is encoded combining S^{-1}(s), 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 including an element into the list of non-zero values, if not present. The operation has therefore a complexity of O (nnz(M)), and (nnz(v)) for vectors.

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

Hadamard Product. The Hadamard product of two vectors u ⊙ v has a complexity of O (nnz(u) nnz(v)).

Tensor Application. For a suitable vector v, the tensor application on the ℓ-th dimension M_{i,j,k}v, with ℓ ∈ {i, j, k}, has asymptotic complexity of O(nnz(M)), 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, ..., s_l\} 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 ≠ 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_i = s_i\*.

7. RESULTS

We implemented our framework into TENSOR RDF, a C++ system using OpenMPI v1.8 library for answering SPARQL queries over RDF datasets. We performed a series of experiments aimed at...
evaluating the performance of our approach, with the main results detailed in this section.

**Benchmark Environment.** We deployed TENSOR-RDF 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 processes running on 2 hosts. We evaluated the performance of TENSOR-RDF 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/sz/gbh67s9ohbphb9/oEGo-8yui). 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 TENSOR-RDF 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. TENSOR-RDF outperforms all competitors, in particular RDF-3X by a large margin. TENSOR-RDF 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 TENSOR-RDF, 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 than Trinity-RDF for LUBM-4450, while 100 times better than MR-RDF-3X and 1.5 times better than Trinity-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 DDF 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.
We also performed warm-cache experiments, but we are unable to report them due to space constraints; however while TENSOR-RDF improves performance from milliseconds to microseconds (e.g., from 1 ms to 0.1 μs), the other competitors improve performance in milliseconds magnitude (e.g., from 100 ms to 1 ms). As last experiment, we tested TENSOR-RDF’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 $10^3$ ms for the largest dimension, i.e., for a number of triples of $10^9$.

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 $2|P| + |S| + |O|$ 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, TENSOR-RDF 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