Querying Compressed Knowledge Bases in Pervasive Computing

Eugenio Di Sciascio[1], Michele Ruta[1], Floriano Scioscia[1], and Eufemia Tinelli[2]

[1] Politecnico di Bari, Via Re David 200, I-70125 Bari, Italy,
`[disciascio, m.ruta, f.scioscia]@poliba.it`,
[2] Universitá degli Studi di Bari, Via Orabona 4, I-70125 Bari, Italy,
`tinelli@di.uniba.it`

Abstract. In the so-called Semantic Web of Things (SWoT), annotated information is tied/derived to/from micro-devices, such as RFID tags and wireless sensors, deployed in an environment. Compression techniques are so needed, due to the verbosity of semantic XML-based languages. Beyond compression ratio, query efficiency is a key aspect for knowledge discovery in mobile ad-hoc scenarios where resources are constrained and topology is unpredictable. This paper proposes a querying schema for OWL knowledge bases, serialized in RDF/XML syntax and homomorphically compressed. The final goal is to allow query evaluation without requiring decompression. Algorithms are presented to prove the feasibility of the proposed approach, while practical examples highlight its usefulness.

1 Introduction and Motivation

In pervasive computing, several factors make information availability as unpredictable: network topology evolution due to node mobility, range limitations and inherent unreliability of wireless communications, node failure due to energy depletion. Thus, approaches based on centralized information storage and management are impractical. Several proposals for collaborative, dynamic resource discovery in ad-hoc networks can be found in literature. In some of them the exploitation of semantics allows to enhance retrieval effectiveness also coping with volatility and unpredictability. The so-called *Semantic Web of Things* (SWoT) aims at the integration of Semantic Web and pervasive computing technologies, in order to associate semantically rich information to real-world objects, locations and events. Data is derived and/or carried by inexpensive, disposable and unobtrusive micro-devices, such as *Radio Frequency IDentification* (RFID) tags and wireless sensors, attached to everyday items or deployed in given environments. Due to power, size and cost constraints, they are usually equipped with little or no processing capabilities, very small storage and short-range, low-throughput wireless links. Data should be extracted and processed by agents on mobile computing devices, through wireless ad-hoc networks.

According to Linked Data best practices, information resources in the Semantic Web should be denoted by dereferenceable URIs (Uniform Resource Identifiers) and annotated in RDF (Resource Description Framework, http://www.w3.

org/TR/rdf-primer/) w.r.t. an RDF Schema (http://www.w3.org/TR/rdf-schema/) or OWL (Web Ontology Language, http://www.w3.org/TR/owl2-overview/) ontology [1]. Language specifications include a standard XML serialization syntax. Adopted knowledge representation models are grounded on formal, logic-based semantics. Query languages, *e.g.*, SPARQL (SPARQL Protocol And RDF Query Language, http://www.w3.org/TR/rdf-sparql-query/), are defined to extract and combine asserted information, while reasoning engines can perform automated inference of knowledge entailed by a given *Knowledge Base* (KB).

One of the most important issues restraining a coherent development of the SWoT vision is the verbosity of the adopted XML-based languages: it is a significant hindrance to efficient storage and transmission of semantic annotations, so that compression techniques become essential. Furthermore, when evaluating encoding algorithms from the SWoT perspective, traditional metrics such as compression ratio and speed are not enough: efficiency and effectiveness of queries on compressed data are critical aspects. Particularly, compression schemes allowing to directly evaluate queries on encoded annotations, without full decompression [2, 3], can be very useful. Unfortunately, so far research has been focused on data extraction from generic structured documents referred to an XML Schema. Hence, main motivation for the present work is that efficient execution of semantic-based queries on compressed KB fragments (ontology segments or resource annotations) can significantly enhance resource discovery capabilities in mobile contexts. So this paper introduces a formal framework for querying OWL Knowledge Bases, serialized in RDF/XML and encoded with *COX (Compressor for Ontological XML-based languages)* [4], which exploits the *homomorphism* property to preserve document structure during compression. Algorithms are defined for the execution of most common elementary queries in the Semantic Web literature. Main contribution is in demonstrating both feasibility and soundness of such general-purpose semantic-based interrogations for on-the-fly knowledge extraction from compressed KBs. Several possibilities are opened for further applied research, devoted to support the integration in high-level query languages (*e.g.*, SPARQL) and inference services by simply and properly combining the proposed query building blocks.

The remaining of the paper is structured as follows. Technical background about KB compression is briefly recalled, before discussing the proposed approach in Section 2, while a case study in Section 3 provides a toy example to highlight usefulness of the proposal. After reporting on relevant related work in Section 4, Section 5 closes the paper.

2 Framework

2.1 Background

Lossless compression is based on substitution of symbols in the input message with code words, according to a statistical model for the input source. *Huffman coding* and *arithmetic encoding* [5] are two fundamental techniques. The former exploits a code table (the *huffman tree*), derived from symbol frequencies.

Widespread universal compression tools *gzip* (http://www.gzip.org/) and *bzip2* (http://bzip.org/) combine Huffman coding with pre-processing input transformations (Lempel-Ziv LZ77 algorithm and Burrows-Wheeler transform, respectively); the latter has better compression rate but it is also slower. In arithmetic encoding, instead, the whole message is represented by a real number in the $[0, 1)$ interval. Disjoint sub-intervals are assigned to all possible symbols, whose length is proportional to symbol frequency. An input message is then mapped to an interval I as follows: at the beginning, $I = [0, 1)$; for each symbol S, I is reduced proportionally to the sub-interval of S. At the end, any value in I will unambiguously encode the sequence of symbols in the message.

Since Semantic Web languages are based on XML syntax, XML-specific algorithms can achieve higher compression rates than universal ones, by exploiting inherent syntactic constraints. An important property is *homomorphism* [2]: homomorphic compression preserves the structure of the original XML data. That may allow to evaluate queries directly on compressed formats, by detecting document pieces which satisfy given query conditions without preliminary decompression of the whole document. *XGrind* [2] and *XPress* [3] are relevant homomorphic compression approaches, adopting Huffman coding and *Reverse Arithmetic Encoding* (RAE), respectively. They achieve high query performance for compressed XML data by virtue of homomorphism; compression rates, though, are lower than the best non-homomorphic XML compression algorithms.

COX (Compressor for Ontological XML-based languages) [4] is adopted here as reference format for querying compressed XML-based semantic annotations. It exploits different solutions to encode data structures (XML tags, attributes) and data (attribute values), in a two-step compression process. For data structures, a RAE variant is used. For attribute values, a dictionary is used to map the most frequent strings to 1-byte codes. COX deals with tag and attribute names in the same way. Attributes are distinguished by a "`@`" prefix. Therefore, in the rest of the section the word "tag" will refer equivalently to a tag or to an attribute.

In the first step, the XML document is parsed and statistics are gathered. After parsing, frequency of each tag name is computed as ratio between the number of occurrences of the tag itself and the total document tags. The $[d, D) = [1.0 + 2^{-7}, 2.0 - 2^{-15})$ interval is then split in disjoint sub-intervals, assigning slightly longer sub-intervals to very rare tags while preserving proportionality with respect to frequency. That avoids encoding errors for tags with a very low frequency. All values representing opening tags fall in the interval $[d, D)$. The interval $[1.0, d)$ is reserved to encode closing tags. Since every possible value is strictly between 1.0 and 2.0, the first byte will always be $\texttt{01111111}_2$ in 32-bit floating point representation, so it can be truncated without loss of information [3]. After the first step a *tag header* is written at the beginning of the output file. It contains a sequence of records composed by: 1 byte for the length of the tag name; the tag name; 3 bytes (after truncation) for encoding the minimum value of the sub-interval related to the tag. The statistic collection for text string frequencies is performed concurrently with the analysis of document structure: strings with both length and frequency higher than heuristic thresholds are encoded. At the end of the first step, a *value header* is written after the tag header. It consists of a sequence of strings, separated by the `ff`$_h$ character. The corresponding codes are single-byte values from `00`$_h$ to `fd`$_h$ and they are assigned to strings in progressive order, hence they can be omitted in the header.

In the second step, the body of the output file is produced. Opening and closing tags, attributes and attribute values are encoded in the same order as they appear in the input document to preserve homomorphism. Each tag T is encoded by applying RAE: as input message, the sequence of tag names is considered, starting with T and going toward its ancestors (hence the adjective "reverse") up to the root XML tag. The sub-interval corresponding to this tag path (named *simple-path*) is computed; then its minimum limit is represented as a 32-bit floating point value and the two central bytes are taken to encode T (the loss of precision due to discarding the least significant byte of the mantissa does not prevent a correct tag identification). Finally, an attribute value is processed as follows: if it belongs to the dictionary produced in the first step, it is replaced by its 1-byte code followed by the delimiter `fe`$_h$, otherwise the string is copied to output, followed by the delimiter `ff`$_h$.

2.2 Querying Approach

In the proposed framework, the classical KB definition $K = \langle \mathcal{T}, \mathcal{A} \rangle$ is adopted, where the TBox $\mathcal{T}$ refers to the ontological knowledge, and the ABox $\mathcal{A}$ specifies the assertional one. The framework deals with KBs in an OWL-DL subset whose characteristics are:

– $\mathcal{T}$ is a *simple TBox*, *i.e.*, a set of *Primitive Concept Specifications* ($A \sqsubseteq B$);
– object properties, data properties and disjoint concepts sets can be defined;
– $\mathcal{A}$ is a role-free ABox, *i.e.*, it is a finite set of individuals defined as instances of a general concept expression C without binary relations between individuals. C can be a conjunction of atomic concepts, unqualified existential quantifications, number restrictions and universal quantifications.

The adoption of a role-free ABox allows to reduce reasoning on assertional knowledge to reasoning on ontological one. Moreover, the selected OWL-DL subset ensures a good trade-off between expressiveness and computational complexity in real applications, as discussed in [6]. Both keyword-based search and a set of path-based queries will be proposed, in order to obtain useful classical inferences on $\mathcal{T}$ and query answering on $\mathcal{A}$. In what follows, it will be shown how, starting from a minimum query set, several other queries can be incrementally built. In fact, the proposed querying approach can be used to cope with non-standard inference services in [7, 6]; deeper discussion is beyond the scope of this work. With reference to TBox reasoning, a set of path-based queries is presented, most of which are exploited in [8]:

– $parents(A)$ - it retrieves all the concepts B such that A is direct sub-class of B. Obviously, it is possible to retrieve all the *ancestors* B of A recursively applying the *parent* primitive to B until B is different from *Top* concept.

```
<rdf:Description rdf:about="http://sisinflab.poliba.it/food.owl#Cavendish">
  <rdf:type>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="http://sisinflab.poliba.it/food.owl#Banana"/>
        <rdf:Description rdf:about="http://sisinflab.poliba.it/food.owl#Ripe_Product"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="http://sisinflab.poliba.it/food.owl#Has_Colour"/>
          <owl:someValuesFrom rdf:resource="&owl;Thing"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="http://sisinflab.poliba.it/food.owl#Has_Colour"/>
          <owl:allValuesFrom rdf:resource="http://sisinflab.poliba.it/food.owl#Yellow"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="http://sisinflab.poliba.it/food.owl#Storage_Humidity"/>
          <owl:someValuesFrom rdf:resource="&owl;Thing"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="http://sisinflab.poliba.it/food.owl#Storage_Humidity"/>
          <owl:allValuesFrom rdf:resource="http://sisinflab.poliba.it/food.owl#Medium_Humidity"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="http://sisinflab.poliba.it/food.owl#Storage_Temperature"/>
          <owl:someValuesFrom rdf:resource="&owl;Thing"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="http://sisinflab.poliba.it/food.owl#Storage_Temperature"/>
          <owl:allValuesFrom rdf:resource="http://sisinflab.poliba.it/food.owl#Room_Temperature"/>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </rdf:type>
</rdf:Description>
```

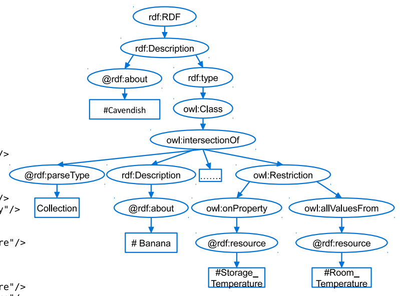

Fig. 1. Instance description in OWL and graph-based representation

- *children*(A) - it retrieves all the concepts B such that B is direct sub-class of A. Also in this case it is possible to retrieve all the *descendants* B of A applying recursively the *children* primitive to B until B is different from *Bottom* concept.
- *properties*(A) - it retrieves all the properties P such that A is domain of P.
- *leaves*(A) - it retrieves all the concepts B as most specific of A. More formally, $leaves(A) = \{B | B = subClassOf(A) \wedge \neg \exists B' : (B' = subClassOf(A) \wedge B' = subClassOf(B))\}$.

With reference to ABox reasoning, two query models are presented: (i) *entity-based search*, implemented by means of string matching on the required concepts and their descendants, and (ii) *path-based queries*. The former is useful when the domain knowledge organization is unknown. The latter can be considered as a solution to the classical *query answering* problem. Note that a KB instance can be seen as graph partition reflecting the representational model used in COX algorithm (see the example in Figure 1). Some nodes can be distinguished as having the same depth w.r.t. document root (*i.e.*, `rdf:RDF` tag), and some other ones are in "partial order" among them. Hence, with reference to the graph in Figure 1, `owl:Restriction` tag directly precedes `owl:allValuesFrom` tag (*i.e.*, one hop of distance), while `owl:allValuesFrom` and `owl:onProperty` tags are at the same depth level.

The proposed query engine refers to the RDF/XML serialization recommended by OWL 2 language specifications. The support for all syntactic variants of RDF/XML is not explicitly dealt with.

Primitives. The following simple-paths will be referenced in query execution algorithms to find elements in the RDF model. For reader's convenience, they are not reported in reverse order.

P1 $rdf : RDF \rightarrow owl : Class \rightarrow @rdf : about$
P2 $rdf : RDF \rightarrow owl : ObjectProperty \rightarrow @rdf : about$
P3 $rdf : RDF \rightarrow owl : Class \rightarrow rdfs : subClassOf \rightarrow @rdf : resource$
P4 $rdf : RDF \rightarrow owl : ObjectProperty \rightarrow rdfs : domain \rightarrow @rdf : resource$
P5 $rdf : RDF \rightarrow rdf : Description \rightarrow @rdf : about$
P6 $rdf : Description \rightarrow rdf : Type \rightarrow owl : Class \rightarrow owl : IntersectionOf \rightarrow owl : Restriction \rightarrow owl : onProperty \rightarrow @resource$

P7 $rdf : Description \rightarrow rdf : Type \rightarrow owl : Class \rightarrow owl : IntersectionOf \rightarrow$
$owl : Restriction \rightarrow owl : onProperty \rightarrow owl : allValuesFrom \rightarrow$
$owl : Class \rightarrow @rdf : about$

P8 $rdf : Description \rightarrow rdf : type \rightarrow owl : Class \rightarrow owl : intersectionOf$
$\rightarrow rdf : Description \rightarrow @rdf : about$

Query execution is based on a set of primitives for accessing COX compressed documents, whose structure, as said, consists of a tag header $\mathcal{H}_T$, a value header $\mathcal{H}_V$ and a body $\mathcal{B}$. The primitives are listed in Table 1 and explained hereafter. $\mathcal{H}_T$, $\mathcal{H}_V$, $\mathcal{B}$ are supposed to be always accessible. Data complexity characterization is provided, along with required (read-only) accesses w.r.t. input size.

- *lookupTag* searches for a tag name in $\mathcal{H}_T$; if found, it returns its associated interval, else it returns *null*.
- *lookupValue* searches for a string value in $\mathcal{H}_V$; if found, it returns its associated 1-byte code, else it returns the value of the input argument.
- *lookupValueLike* searches for $\mathcal{H}_V$ within strings containing the input argument; it returns the (possibly empty) set of 1-byte codes associated to matching strings.
- *lookupCode* searches for a code in $\mathcal{H}_V$; if found, it returns its associated string, else it returns *null*.
- *computeSimplePath* computes the simple path interval; it uses the arithmetic encoding algorithm described in Section 2.1 and requires a *lookupTag* call for each element in the simple path.
- *findPathsWithValue* takes in input an interval i and a string value v; it gets $c := lookupValue(v)$, then it scans $\mathcal{B}$ to find all occurrences of the simple path encoded by i followed by c; they are returned as positions (in bytes) from the $\mathcal{B}$ beginning.
- *findPathsWithValueLike* is similar to the previous primitive. It takes in input an interval i and a string value v, and it scans $\mathcal{B}$ to find all occurrences of the simple path encoded by i followed by a string containing v; they are returned as positions (in bytes) from the $\mathcal{B}$ beginning.
- *getValuesAfter* takes in input an interval i and a position n; it scans the document from position n, up to the end of the related XML element. It returns a (possibly empty) set of string values following attributes encoded with a value in i.
- *getValuesBefore* scans the document backwards from position n up to the beginning of the nth XML element.

Queries. Algorithm 1 and Algorithm 2 exploit simple-paths $P1$ and $P3$ and COX access primitives to find parents and children of a given class, respectively. Algorithm 3 (resp. 4) calls Algorithm 1 (resp. 2) to find the class ancestors (resp. descendants). In order to find leaves of a given class the previous algorithms have to be exploited, as reported in Algorithm 5. Algorithm 6 uses simple-path $P4$ to look up for a domain relationship between the input class and a property name, then $P2$ to find the property name by scanning the compressed document backwards. Algorithm 7 performs keyword-based search using partial string matching both in the document header and body. Finally, Algorithm 8 finds the ABox individuals that are instances of a class intersection.

Name	Input	Output	Complexity
$lookupTag(t)$	tag name t	interval or *null*	$O(\lvert\mathcal{H}_T\rvert)$
$lookupValue(v)$	string value v	code of v or v itself	$O(\lvert\mathcal{H}_V\rvert)$
$lookupValueLike(v)$	string value v	(possibly empty) set of codes of strings containing v	$O(\lvert\mathcal{H}_V\rvert)$
$lookupCode(c)$	code c between 0 and 253	string at position c in $\mathcal{H}_V$ or *null*	$O(\lvert\mathcal{H}_V\rvert)$
$computeSimplePath(P)$	simple path P	interval or *null*	$O(\lvert P\rvert \times \lvert\mathcal{H}_T\rvert)$
$findPathsWithValue(i, v)$	interval i of simple path, string value v	(possibly empty) set of occurrences, as positions from start of document	$O(\lvert\mathcal{B}\rvert + \lvert\mathcal{H}_V\rvert)$
$findPathsWithValueLike(i, v)$	interval i of simple path, string value v	(possibly empty) set of occurrences, as positions from start of document	$O(\lvert\mathcal{B}\rvert)$
$getValuesAfter(i, n)$	interval i, position n	(possibly empty) set of strings	$O(\lvert\mathcal{B}\rvert + \lvert\mathcal{H}_V\rvert)$
$getValuesBefore(i, n)$	interval i, position n	(possibly empty) set of strings	$O(\lvert\mathcal{B}\rvert + \lvert\mathcal{H}_V\rvert)$

Table 1. Access primitives for a COX compressed document

Algorithm 1 *parents*(a)

```
Require: $a$ class name, $P1$ and $P3$ simple-paths
Ensure: $P$ set of parents of $a$
1: $P := \emptyset$
2: $i_1 := computeSimplePath(P1)$
3: $i_2 := computeSimplePath(P3)$
4: $N := findPathsWithValue(i_1, a)$
5: for all $n \in N$ do
6:     $P := P \cup getValuesAfter(i_2, n)$
7: end for
```

Algorithm 2 *children*(a)

```
Require: $a$ class name, $P1$ and $P3$ simple-paths
Ensure: $C$ set of children of $a$
1: $C := \emptyset$
2: $i_1 := computeSimplePath(P3)$
3: $i_2 := computeSimplePath(P1)$
4: $N := findPathsWithValue(i_1, a)$
5: for all $n \in N$ do
6:     $C := C \cup getValuesBefore(i_2, n)$
7: end for
```

Algorithm 3 *ancestors*(a)

```
Require: $a$ class name
Ensure: $A$ set of ancestors of $a$
1: $A := \emptyset$
2: $P := parents(a)$
3: $A := P$
4: for all $p \in P$ do
5:     $A := A \cup ancestors(p)$
6: end for
```

Algorithm 4 *descendants*(a)

```
Require: $a$ class name
Ensure: $D$ set of descendants of $a$
1: $D := \emptyset$
2: $C := children(a)$
3: $D := C$
4: for all $c \in C$ do
5:     $D := D \cup descendants(c)$
6: end for
```

Algorithm 5 *leaves*(a)

```
Require: $a$ class name
Ensure: $L$ set of leaves of $a$
1: $L := \emptyset$
2: $C := children(a)$
3: if $C == \emptyset$ then
4:     $L := L \cup \{a\}$
5: else
6:     for all $c \in C$ do
7:         $L := L \cup leaves(c)$
8:     end for
9: end if
```

Algorithm 6 *properties*(a)

```
Require: $a$ class name, $P2$ and $P4$ simple-paths
Ensure: $P$ list of properties having $a$ as domain
1: $P := \emptyset$
2: $i_1 := computeSimplePath(P4)$
3: $i_2 := computeSimplePath(P2)$
4: $C := ancestors(a) \cup a$
5: for all $c \in C$ do
6:     $N := findPathsWithValue(i_1, c)$
7:     for all $n \in N$ do
8:         $P := P \cup getValuesBefore(i_2, n)$
9:     end for
10: end for
```

3 Case Study

In order to clarify how the proposed framework works and how the different query models can be used, a simple case study in RFID supply chain management is

Algorithm 7 $keyword-based\_search(A_1, \ldots, A_n)$

```
Require: $a_1, \ldots, a_n$ names to search, $P1$ simple path
Ensure: $C$ set of classes syntactically similar to $a_1, \ldots, a_n$
 1: $C := \emptyset$
 2: $i := computeSimplePath(P1)$
 3: for $k = 1$ to $n$ do
 4:     $V := lookupValueLike(a_k)$
 5:     for all $v \in V$ do
 6:         if $v! = null$ then
 7:             $n := findPathsWithValue(i, v)$
 8:             if $n! = \emptyset$ then
 9:                 $C := C \cup \{lookupCode(v)\}$
10:             end if
11:         else
12:             $n := findPathsWithValueLike(i, v)$
13:             if $n! = \emptyset$ then
14:                 $C := C \cup \{v\}$
15:             end if
16:         end if
17:     end for
18: end for
```

Algorithm 8 $entity-based\_search(a_1, \ldots, a_n)$

```
Require: $a_1, \ldots, a_n$ classes names, $P5$ and $P8$ simple paths
Ensure: $Ins$ list of individuals instance of the intersection of $a_1, \ldots, a_n$
 1: $Ins := \emptyset$
 2: $i_1 := computeSimplePath(P8)$
 3: $i_2 := computeSimplePath(P5)$
 4: for $1 = 1$ to $n$ do
 5:     $C_i := descendants(a_i) \cup \{a_i\}$
 6:     for all $c \in C_i$ do
 7:         $N := findPathsWithValue(i_1, c)$
 8:         for all $n \in N$ do
 9:             $Ins_i := Ins_i \cup getValuesBefore(i_2, n)$
10:         end for
11:     end for
12: end for
13: $Ins := Ins_1 \cap \ldots \cap Ins_n$
```

considered, where each RFID tag stores a compressed semantic annotation of the product/stock it is attached to. In [6], backward-compatible extensions of EPCglobal RFID tag data structure were devised to accommodate an RDF product description, that could be read by means of standard RFID reader-tag air interface protocol. However, the case study concerns semantic-based queries upon KB in compressed COX format at a logical level, thus it can be applied to any physical data storage model.

Let us consider a large distribution warehouse, receiving several kinds of products manufacturers. A truck fleet is used to deliver products toward sale points. Each truck is endowed with a mobile computing device with embedded semantic-enabled RFID reader and on-board query processing capabilities. When a product is loaded into the truck storage compartment, the reader extracts the compressed annotation from the RFID tag of the product, and queries it in order to check: (i) if the product type is compatible with those the truck is allowed to transport and (ii) if characteristics of the storage compartment are adequate for the product (*e.g.,* lighting, humidity, temperature and any special storage or security equipments). Any incompatibility would likely indicate an error in truck assignment or product routing within the warehouse, hence it must be discovered immediately, through on-the-fly semantic query processing.

A stock S of Cavendish bananas is loaded on a truck T, which is allowed to transport fruit. Storage compartment of T provides no thermostat, hence it can only keep products at room temperature. S has an RFID tag with the compressed semantically annotated product description, expressed w.r.t. a suitable product ontology. Let us suppose that the product annotation coincides with Figure 1 and that the following assertions are in the reference ontology; Notation3 (http://www.w3.org/DesignIssues/Notation3.html) is adopted here for reader's convenience:

Algorithm 9 *checkUniversalRestriction*(p, f)

```
Require: $p$ property, $f$ atomic filler, $P6$, $P7$ simple-paths
Ensure: return $true$ if the individual contains a $\forall p.f$ restriction, $false$ otherwise
 1: $i_1 := computeSimplePath(P6)$
 2: $i_2 := computeSimplePath(P7)$
 3: $D := descendants(f) \cup \{f\}$
 4: $N := findPathsWithValue(i_1, p)$
 5: for all $n \in N$ do
 6:    $V := getValuesAfter(i_2, n)$
 7:    for all $v \in V$ do
 8:       if $v \in D$ then
 9:          return $true$
10:       end if
11:    end for
12: end for
13: return $false$
```

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix o: <http://sisinflab.poliba.it/food.owl#> .
o:Banana rdfs:subClassOf o:Fruit .
o:Uncontrolled_Temperature rdfs:subClassOf o:Temperature .
o:Room_Temperature rdfs:subClassOf o:Uncontrolled_Temperature .
o:Storage_Temperature rdfs:range o:Temperature .
```

Entity-based and keyword-based search - According to Algorithm 8, the computing device embedded in the truck runs $entity - based\_search(Fruit)$ on the description of $S.\ Cavendish$ is described as an instance of $Banana$, which is subclass of $Fruit$, hence it is returned. It is useful to point out that a keyword-based search can support the selection of suitable ontology classes before an entity-based search.

Path-based search - When known the KB structure and organization, it is possible to compose more complex queries for instance retrieval. In the above example, the device embedded in the truck has to check if the product description is compliant with the $\forall Storage\_Temperature.Uncontrolled\_Temperature$ restriction. To do so, Algorithm 9 is executed with $p = Storage\_Temperature$ and $f = Uncontrolled\_Temperature$. Intervals for simple-paths are computed first, and the set D of descendants of f is extracted from the TBox. Then the universal restriction is searched in line 4: if it is found and its filler belongs to D, the constraint is satisfied and function returns $true$. Notice that expanding the search for descendants of the filler, makes the check semantic-based rather than purely syntactic. In the example, checking the individual in Figure 1 returns a positive answer, because $Room\_Temperature$ is a subclass of $Uncontrolled\_Temperature$. Other constraints can be verified in a similar fashion; as an optimization, computed intervals for simple-paths can be cached for reuse during query processing on the same compressed RDF annotation.

A partial implementation of the query engine has been performed, including the primitives in Table 1. Experiments were executed on a notebook with Intel Core 2 Duo CPU (2.0 GHz clock frequency), 3 GB RAM and Ubuntu 10.04 OS (Linux 2.6.32 kernel version). Algorithm 9 was tested on a KB with 175 concepts, 11 roles and 15 instances (original OWL size is 114 kB, COX compressed size is 31 kB). Average execution time was 460 ms and main memory usage peak was 1.51 MB. In a further test, 260 *findPathsWithValue* primitives were executed with pseudo-random arguments: Figure 2 reports the distribution of times. These preliminary results suggest that the approach is viable. Since the implementation is not complete, comparative tests w.r.t. other query engines cannot be reported at this time.

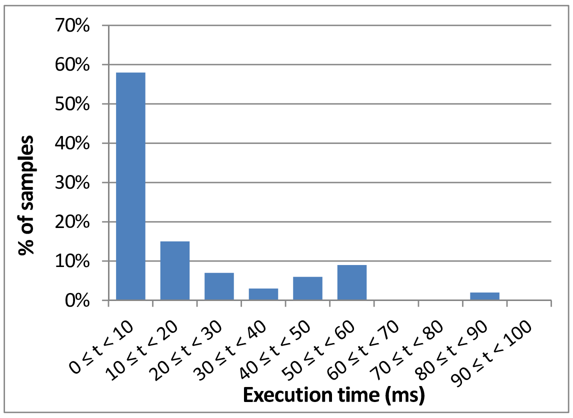

Fig. 2. Execution time of *findPathsWithValue* primitive

4 Related Work

With reference to XML-based languages, several tools supporting efficient querying over compression schemes exist. *XGrind* [2] can perform (i) exact-match and prefix-match queries directly on compressed values and (ii) range and partial-match queries on values decompressed on-the-fly. *XPress* [3] exploits RAE to improve the path-based queries. XPress query engine converts a label path expression into a sequence of intervals. Then, by using this sequence, the query executor checks whether the encoded values of XML tags are in a given interval of the sequence or not. *XQueC* [9] exploits indexing and XML storage strategies since it is focused on search speed rather than compression efficiency. The above tools execute path-based queries expressed in XPath (http://www.w3.org/TR/xpath/), which allow syntactic match of document elements and are strictly tied to the XML Schema for the particular document. Therefore, it is not possible to reuse an existing approach for semantic-based queries.

In known strategies for storing and querying RDF annotations, data structures and optimizations are focused on a database perspective [8]. The Semantic Web community has generally used traditional database systems [10]. As a consequence, most of the RDF query processing techniques are based on database query processing and optimization techniques, mainly focused on compression [11] and indexing approaches [12, 13]. Nevertheless, all these technologies do not cope with mobile computing issues. An interesting exception is the MQuery [14] framework for ubiquitous computing. It creates a compressed index of RDF graphs for improving context-aware retrieval, according to the idea that a mobile user wants to access specific data depending on given situations. The main drawback w.r.t. the approach proposed here is the limited flexibility and expandability, as MQuery provides a pre-defined query interface for selecting only from four possible interrogations.

Studies on the above works have suggested main query models expected by semantic-based applications: (i) full-text search, *i.e.*, keyword or string matching; (ii) queries based on data structure, *i.e.*, path-based and structure-based queries; (iii) a combination of them. As a consequence, the presented framework included both keyword-based search and a set of path-based queries.

5 Conclusion

A framework has been presented for querying knowledge bases expressed in OWL, serialized in RDF/XML and processed with a homomorphic compression. It is particularly suitable for scenarios dipped in the Semantic Web of Things vision. Primitives for querying compressed semantic annotations have been devised and used to perform interrogations on both the TBox and KB instances. The implementation of the framework in a software tool is ongoing. It will provide the needed insight into performance benefits and costs, allowing to evaluate possible optimizations. Moreover, it will support the development of a semantic-based query and reasoning engine for compressed KBs, by exploiting basic queries to implement high-level query languages and inference services commonly used in the Semantic Web.

References

1. Bizer, C., Heath, T., Berners-Lee, T.: Linked data-the story so far. Int. J. Semantic Web Inf. Syst. **5**(3) (2009) 1–22
2. Tolani, P., Haritsa, J.: XGRIND: A Query-friendly XML Compressor. In: Proc. of the 18th Int. Conf. on Data Engineering (ICDE.02), IEEE (2002) 225234
3. Min, J., Park, M., Chung, C.: A compressor for effective archiving, retrieval, and updating of XML documents. ACM Transactions on Internet Technology **6**(3) (2006) 223–258
4. Scioscia, F., Ruta, M.: Building a Semantic Web of Things: issues and perspectives in information compression. In: Semantic Web Information Management (SWIM'09). In Proc. of the 3rd IEEE Int. Conf. on Semantic Computing (ICSC 2009), IEEE Computer Society (2009) 589–594

5. Witten, I., Neal, R., Cleary, J.: Arithmetic coding for data compression. Communications of the ACM **30**(6) (1987) 520–540
6. Di Noia, T., Di Sciascio, E., Donini, F.M., Ruta, M., Scioscia, F., Tinelli, E.: Semantic-based Bluetooth-RFID interaction for advanced resource discovery in pervasive contexts. Int. Jour. on Semantic Web and Information Systems **4**(1) (2008) 50–74
7. Ruta, M., Zacheo, G., Grieco, L.A., Di Noia, T., Boggia, G., Tinelli, E., Camarda, P., Di Sciascio, E.: Semantic-based Resource Discovery, Composition and Substitution in IEEE 802.11 Mobile Ad Hoc Networks. Wireless Networks **16**(5) (2010) 1223–1251
8. Christophides, V., Plexousakis, D., Scholl, M., Tourtounis, S.: On labeling schemes for the Semantic Web. In: The 12th Int. Conf. on World Wide Web, ACM (2003) 544–555
9. Skibiski, P., Swacha, J.: Combining Efficient XML Compression with Query Processing. In: Advances in Databases and Information Systems. Volume 4690. Springer Berlin / Heidelberg (2007) 330–342
10. Sakr, S., Al-Naymat, G.: Relational processing of RDF queries: a survey. SIGMOD Rec. **38** (June 2010) 23–28
11. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix Bit loaded: a scalable lightweight join query processor for RDF data. In: The 19th Int. Conf. on World Wide Web, ACM (2010) 41–50
12. Delbru, R., Toupikov, N., Catasta, M., Tummarello, G.: A node indexing scheme for web entity retrieval. In: The Semantic Web: Research and Applications. Volume 6089. (2010) 240–256
13. Zhang, S., Yang, J., Jin, W.: SAPPER: Subgraph Indexing and Approximate Matching in Large Graphs. Proc. of the VLDB Endowment **3**(1) (2010) 1185–1194
14. Zhang, Y., Zhang, N., Tang, J., Rao, J., Tang, W.: Mquery: Fast graph query via semantic indexing for mobile context. In: The 2010 IEEE/WIC/ACM Int. Conf. on Web Intelligence and Intelligent Agent Technology - Volume 01, IEEE Computer Society (2010) 508–515