# A Framework for Query Processing over Compressed Knowledge Bases

Floriano Scioscia
Dipartimento di Elettrotecnica ed Elettronica
Politecnico di Bari
Via Re David 200, I-70125, Bari, Italy
Email: f.scioscia@poliba.it

Eufemia Tinelli
Dipartimento di Informatica
Università di Bari
Via Orabona, I-70125, Bari, Italy
Email: tinelli@di.uniba.it

***Abstract*—In semantic-based pervasive computing, annotated information is tied to micro-devices, such as RFID tags and wireless sensors, deployed in an environment. Compression techniques are essential, because of the verbosity of standard XML-based languages for ontologies and semantic annotations. Beyond compression ratio, query efficiency is a key aspect. This paper presents a framework for querying knowledge bases expressed in OWL, serialized in RDF/XML syntax and compressed with a homomorphic encoding, in order to allow query evaluation without requiring decompression. Formalization of a significant set of queries demonstrates feasibility of the approach, while practical examples highlight its usefulness.**

## I. INTRODUCTION

In the Semantic Web, available information resources should be annotated in RDF (Resource Description Framework, http://www.w3.org/TR/rdf-primer/), with respect to an ontology in RDF Schema or OWL (Web Ontology Language, http://www.w3.org/TR/owl2-overview/) defining a common vocabulary for a domain. Language specifications include a standard XML serialization syntax. Query languages, such as SPARQL (http://www.w3.org/TR/rdf-sparql-query/), are defined to extract and combine asserted information, while reasoning engines (based on OWL-DL subset) can perform automated inference of knowledge entailed by a given *Knowledge Base* (KB) of asserted concepts, relationships and facts.

The integration of Semantic Web and pervasive computing technologies aims at associating semantically annotated information with real-world objects, locations and events, through micro-devices capable of sensing and/or carrying useful data; notable technologies include *Radio Frequency IDentification* (RFID) and *wireless sensor networks*. Such data should be automatically extracted and processed by agents on mobile computing devices, through Mobile Ad-hoc Networks (MANETs), in order to better support current user activities.

In pervasive contexts, events cause information needs that must be satisfied immediately. Furthermore, several factors make information resource availability unpredictable: mobility of objects and nodes, range limitations and inherent unreliability of wireless communications, node failure due to energy depletion. The capability of exploiting volatile resources to satisfy immediate needs is often defined as *opportunistic* networking/computing: traditional approaches, based on centralized information storage and management, are evidently impractical. Furthermore, XML-based languages adopted in the Semantic Web are too verbose to allow efficient data management. Compression techniques become essential, in order to enable storage and transmission of semantically annotated information on micro-devices such as RFID tags or wireless sensors. When evaluating encoding algorithms from an information management standpoint, traditional metrics such as compression ratio and speed do not provide the full picture. Efficiency of queries on compressed data becomes a critical parameter, even more so in ad-hoc contexts.

This paper presents a formal framework for querying KB fragments expressed in OWL, serialized in RDF/XML syntax and encoded with *COX (Compressor for Ontological XML-based languages)* [1], an approach exploiting the *homomorphism* property to preserve XML document structure during compression. Algorithms are defined for the execution of some of the most significant query types adopted in the Semantic Web literature, related to both terminological (TBox) and assertional (ABox) knowledge. The main contribution is demonstrating the feasibility and soundness of a set of general-purpose semantic queries for on-the-fly knowledge extraction from compressed KBs. The theoretical formalization opens possibilities for further applied research, devoted to supporting the implementation of high-level query languages, such as SPARQL, and inference services by combining the proposed building blocks.

The rest of the paper is structured as follows. Section II motivates the work by outlining the main challenges that are faced. Technical background and related work about KB compression and querying are recalled in Section III. Section IV describes the framework in detail, while Section V provides practical query examples. Finally, Section VI closes the paper.

## II. CHALLENGES

The proposed approach is aimed at pervasive computing scenarios characterized by large numbers of inexpensive, disposable and unobtrusive micro-devices (such as RFID tags or wireless sensors) capable of producing and/or carrying semantically annotated data within a given environment. Mobile computing devices extract and exploit such information, by acting as cluster-heads with respect to micro-devices in their direct range and by exploiting MANETs to exchange information with nearby peers. Notable application scenarios of semantic-enhanced RFID solutions are smart supply chain management [2], ubiquitous commerce [3] and healthcare (both in hospitals and for ubiquitous health monitoring). Applications of wireless semantic sensor networks include environmental monitoring (also for road conditions and within buildings), disaster recovery and precision agriculture.

Several technological challenges characterize such contexts: (1) due to strict power, size and cost constraints, micro-devices have little or no processing capabilities, very small storage and short-range, low-throughput wireless links; (2) MANET hosts are battery-powered mobile computing devices, and communication is generally more expensive than computation from an energy standpoint; (3) wireless communications are vulnerable to electromagnetic noise; (4) applications generally need on-the-fly query answering. Compression of semantically annotated information introduces benefits in each of these areas: more information can be stored in limited memory amounts; due to lower data transfer times, power consumption is decreased, the impact of noise is mitigated [3] and query processing latency can be reduced.

Semantic-based resource discovery frameworks for MANETs can be found in literature. In some approaches [2], [4], resources are pre-selected by matching only the reference ontology identifier and ancillary extra-logical, data-oriented attributes w.r.t. the request. This hybrid solution grants efficiency, but semantic data management capabilities are somewhat limited. Other approaches choose a direct reuse of Semantic Web technologies (*e.g.*, [5] uses SPARQL for queries and HTTP protocol for resource access), but significant performance issues can ensue. The main goal of the present work is to provide a feasible approach for the execution of semantic-based queries on compressed KB fragments (ontology segments or resource annotations). That can enhance significantly semantic data management capabilities in pervasive contexts. To this aim, *homomorphic* compression algorithms for XML-based semantic annotations –briefly recalled in the next section– are exploited. Full decompression introduces a fixed processing step that can have a significant performance impact, particularly on relatively simple queries, which can be considered as a frequent case in pervasive scenarios. Therefore, we deem that avoiding full decompression is important to improve efficiency of on-the-fly query processing. Finally, a suitable communication protocol (*e.g.*, extending the ones in [2], [4]) is needed to support semantic requests and replies among MANET hosts, but that is outside the scope of the paper.

## III. Background

### A. COX Compression

In this work, *COX* [1] is adopted as reference format for querying compressed XML-based semantic annotations. COX exploits two different solutions to encode data structures and data, in a two-step compression process. For data structures (XML tags and attributes), a *Reverse Arithmetic Encoding* (RAE) [6] variant is used. For attribute values, a dictionary maps 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 added to the name. Therefore, in the rest of the section the word "tag" will refer equivalently to a tag or an attribute.

In the first step the XML document is parsed and statistics are gathered. After parsing, an adjusted frequency of each tag name is computed as the 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 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 that represent 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 [6]. 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 the encoding of the minimum value of the sub-interval of the tag. The statistics collection for text string frequencies is performed concurrently with the analysis of document structure. At the end of the first step, a *value header* is written after the tag header. It consists in a sequence of strings, separated by the $\texttt{ff}_h$ character. The corresponding codes are single-byte values from $\texttt{00}_h$ to $\texttt{fd}_h$ and they are assigned to strings in progressive order, hence they are 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 (homomorphism). Each tag $t$ is encoded by applying RAE: as input message, the sequence $S = (s_0, s_1, \ldots, s_n)$ of tag names is considered, starting with $s_0 = t$ and going toward its ancestors (hence the adjective "reverse") up to the root XML tag. This tag path (named *simple-path*) is then mapped to an interval $I$ as follows: at the beginning, $I = [d, D)$; for each $s_i$, $I$ is reduced proportionally to the sub-interval of $s_i$; at the end of $S$, the minimum limit of $I$ is represented as a 32-bit floating point value and the two central bytes are taken to encode $T$. 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 $\texttt{fe}_h$, otherwise the string is copied to output, followed by the delimiter $\texttt{ff}_h$.

### B. Related Work

With reference to XML-based languages, several tools supporting efficient querying over compression schemes exists; see [7] for a comprehensive survey. *XGrind* [8] 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* [6] exploits RAE to improve the path-based queries. XPress query engine transforms 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 an 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, which allow syntactic match of document elements and are strictly tied to the XML Schema of the compressed document. Therefore, it is not possible to reuse an existing approach for semantic-based queries, but new query primitives must be defined w.r.t. COX format.

In existing strategies for storing and querying RDF annotations, data structures and optimizations are focused on a database perspective [10]. The Semantic Web community has generally used traditional database systems [11], even though recently alternative technologies have been proposed by *e.g.*, *NoSQL* (http://nosql-database.org/). Moreover, standard RDF query language SPARQL closely follows SQL syntax. Consequently, most of the RDF-based query processing techniques rely on database optimizations [12], [13]. These technologies do not cope with mobile computing issues. An interesting exception is the MQuery [14] framework. 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 certain situations. The main drawback w.r.t. our approach is limited flexibility and extensibility, as MQuery provides a pre-defined query interface for selecting only from four possible query types.

Study of the above works suggested the main query types that semantic-based applications expect: (i) full-text search, *i.e.*, keyword or string matching; (ii) queries based on data structure, *i.e.*, path-based and structural-based queries and (iii) a combination of them. Accordingly, our proposal includes both keyword-based search and a set of path-based queries.

## IV. Framework

### A. Query Model

In the proposed framework, the classical KB definition $K = \langle \mathcal{T}, \mathcal{A} \rangle$ is adopted, where the *TBox* $\mathcal{T}$ specifies the ontological knowledge, and the *ABox* $\mathcal{A}$ specifies the assertional one. The framework deals with KBs in an OWL-DL subset specified as follows: (i) $\mathcal{T}$ is a *simple Tbox*, *i.e.*, a set of *Primitive Concept Specifications* ($A \sqsubseteq B$); (ii) object properties, data properties and disjoint concepts sets can be defined; (iii) $\mathcal{A}$ is a role-free ABox, *i.e.*, 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.

With reference to TBox reasoning, a set of path-based queries is presented (most of which are exploited in [10]):

- $parents(A)$ - it retrieves all the concepts $B$ such that $A$ is direct sub-class of $B$. Obviously, it is possible to retrieve all the ancestors of $A$ by applying recursively the $parent$ primitive.
- $children(A)$ - 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 of $A$ by applying recursively the $children$ primitive.
- $properties(A)$ - it retrieves all the properties $p$ such that $A$ is domain of $p$.
- $leaves(A)$ - it retrieves all the concepts $B$ such that $B$ are the most specific concepts of $A$. More formally, we say that $leaves(A) = \{B | B = subClassOf(A) \wedge \neg\exists B' : (B' = subClassOf(A) \wedge B' = subClassOf(B))\}$.
- $nca(A_1, \ldots, A_n)$ - it retrieves the nearest common ancestor of $n$ concepts. In other words, $nca$ query retrieves the most specific concept of a collection composed of the ancestors common to all the $n$ concepts. More formally, $nca(A) = \{B | A_1 = subClassOf(B) \wedge \ldots \wedge A_n = subClassOf(B) \wedge \neg\exists B' : (A_1 = subClassOf(B') \wedge \ldots \wedge A_n = subClassOf(B') \wedge B' = subClassOf(B))\}$.

```
<rdf:RDF xmlns="http://sisinflab.poliba.it/unnamed.owl#"
<rdf:Description rdf:about="#Temperature_Aid_Kit">
        <rdf:type>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <rdf:Description rdf:about="#First_aid_kit"/>
                    <rdf:Description rdf:about="#Loudspeaker"/>
                    <rdf:Description rdf:about="#Thermometer"/>
                    <owl:Restriction>
                        <owl:onProperty rdf:resource="#measures"/>
                        <owl:someValuesFrom rdf:resource="#Thing"/>
                    </owl:Restriction>
                    <owl:Restriction>
                        <owl:onProperty rdf:resource="#measures"/>
                        <owl:allValuesFrom rdf:resource="#Temperature"/>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </rdf:type>
  </rdf:Description >
</rdf:RDF>
```

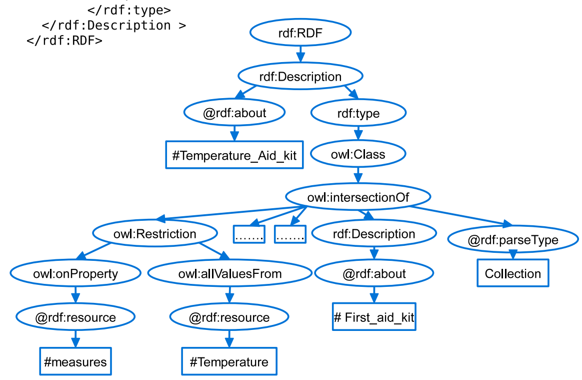


Fig. 1. Instance description in OWL and graph-based COX representation

With reference to ABox reasoning, two types of queries are presented: (i) entity-based search, implemented by means of a string matching on the required concepts and their descendants, and (ii) more complex path-based queries. The former is useful when the knowledge structure is not known. The latter can be considered as a solution to the classical *query answering* problem. In particular, we consider three path-based queries on ABox. COX sees an OWL individual as a tree of nodes; in the example of Figure 1, `owl:Restriction` tag precedes directly `owl:allValuesFrom` (*i.e.*, one hop of distance), whereas `owl:allValuesFrom` and `owl:onProperty` are at the same depth level. The following path-based queries on ABox are needed, since it is not possible to use the ISA relation underlying the path-based queries on the TBox:

– $a//b$: it retrieves all the individuals having a node $a$ which precedes a node $b$. In other words, we retrieve all the individuals having at least a path between $a$ and $b$;

– $a/b$: it retrieves all the individuals having a node $a$ which precedes directly a node $b$;
– $a \leftrightarrow b$: it retrieves all the individuals having nodes $a$ and $b$ at the same depth level.
Keyword-based search on the TBox can be exploited to suggest class names to be used in the query composition.

### B. Query Engine Formalization

The proposed query engine can be formalized now. The approach refers to the constructors listed in Section IV-A, expressed in the RDF/XML serialization recommended by OWL 2 language specifications. The management of all syntactic variants of RDF/XML is not explicitly dealt with in this work.

**Primitives**. The following simple-paths are 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 | $rdf : RDF \rightarrow rdf : Description \rightarrow rdf : type \rightarrow owl : Class \rightarrow @rdf : about$ |
| P7 | $rdf : RDF \rightarrow rdf : Description \rightarrow rdf : type \rightarrow owl : Class \rightarrow owl : intersectionOf \rightarrow rdf : Description \rightarrow @rdf : about$ |

Query execution is based on a set of primitives for the access to a COX compressed document, whose structure, as said, consists in a tag header $\mathcal{H}_T$, a value header $\mathcal{H}_V$ and a body $\mathcal{B}$. The primitives are listed in Table I and explained hereafter. The word *token* will be used to denote any of the following: an opening tag; a closing tag; the start of an attribute; the end of an attribute. $\mathcal{H}_T$, $\mathcal{H}_V$, $\mathcal{B}$ are supposed to be always accessible and therefore are not part of the input. Data complexity characterization is provided, as the number of required (read-only) accesses w.r.t. document size.
– *lookupTag* searches a tag name in $\mathcal{H}_T$; if found, it returns its associated interval, else it returns *null*.
– *lookupValue* searches 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 $\mathcal{H}_V$ for strings containing the input argument; it returns the (possibly empty) set of 1-byte codes associated to matching strings.
– *lookupCode* searches a code in $\mathcal{H}_V$; if found, it returns its associated string, else it returns *null*.
– *computeSimplePath* computes the interval for a simple-path; it uses the arithmetic encoding algorithm described in Section III-A and requires a *lookupTag* call for each element in the simple-path.
– *getNextToken* takes position $n$ and returns the value encoding the next token in $\mathcal{B}$ after position $n$.
– *getPreviousToken* takes position $n$ and returns the value encoding the previous token in $\mathcal{B}$ before position $n$.
– *isStartToken* takes number $n$ encoding a token and returns *true* if it is the start of an XML element or attribute, *false* otherwise.
– *getNextValue* takes position $n$ and returns the next (possibly encoded) attribute value.
– *findSimplePathsWithValue* 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 $i$ and immediately followed by $c$; they are returned as positions (in bytes) from the start of $\mathcal{B}$. This primitive is useful to search for a specific attribute value, which in RDF/XML is needed to find *e.g.*, occurrences of a class name.
– *findSimplePathsWithValueLike* is similar to the previous primitive. It takes in input an interval $i$ and a string value $v$. It scans $\mathcal{B}$ to find all occurrences of the simple-path encoded by $i$ and immediately followed by a string containing $v$; they are returned as positions (in bytes) from the start of $\mathcal{B}$.
– *getValuesAfterPosition* takes in input an interval $i$, and a position $n$; it scans the document from position $n$, up to the end of the XML element at $n$. It returns a (possibly empty) set of string values that immediately follow attributes encoded with a number within $i$. Algorithm 1 formalizes this primitive, which is useful to get values of attributes within a specific XML element.
– *getValuesBeforePosition* is the dual primitive of *getValuesAfterPosition*. It scans the document backwards from position $n$, up to the start of the XML element at $n$.

**TBox queries**. Algorithm 2 and Algorithm 3 exploit simple-paths $P1$ and $P3$ and COX access primitives to find parents and children of a given class, respectively. Algorithm 4 (respectively 5) calls Algorithm 2 (resp. 3) to find the class ancestors (resp. descendants). Finding leaves of a class and the nearest common ancestor of a set of classes exploit the previous algorithms, as reported in Algorithm 6 and 7. Algorithm 8 uses simple-path $P4$ to look up for a domain relationship between the input class and a property name, then $P2$ is used to find the property name by scanning the compressed document backwards. Finally, Algorithm 9 uses partial string matching both in the document value header and body to find the input keyword.

**ABox queries**. Algorithm 10 and 11 allow to find the ABox individuals that are instances of a class and of an intersection of classes, respectively.

## V. Illustrative example

To better clarify the framework and the different query types, a simple example in a *wireless semantic sensor and actor network* is considered. Let us suppose that the action planning to respond to an emergency is performed on-the-fly by a mobile coordinator unit, which must select the best actors for intervention based on their advertised capabilities. The coordinator is able to execute queries on its local TBox copy and to issue ABox queries to other units in order to discover which ones match required characteristics. The example task is to find a unit equipped with an "aid kit" and "sensors" able to measure "weather" conditions. According to a modified version of the ontology developed in [4] in order to satisfy the simple TBox condition (not reported here due to lack of

**Algorithm 1** $getValuesAfterPosition(i, n)$

**Require:** $i$ simple-path interval, $n$ position $\in \{0, \ldots, |\mathcal{B}|\}$
**Ensure:** $V$ set of values

```
pos := n
C = ∅
count := 1
while count > 0 do
    t := getNextToken(pos)
    pos := pos + sizeof(t)
    if isStartToken(t) then
        count := count + 1
        if t ∈ i then
            c := getNextValue(pos)
            C := C ∪ {c}
            pos := pos + sizeof(c)
        end if
    else
        count := count − 1
    end if
end while
for all c ∈ C do
    if c is a string then
        V = V ∪ {c}
    else
        V = V ∪ {lookupCode(c)}
    end if
end for
```

**Algorithm 2** $parents(a)$

**Require:** $a$ class name, $P1$ and $P3$ simple-paths
**Ensure:** $P$ set of parents of $a$

```
P := ∅
i_1 := computeSimplePath(P1)
i_2 := computeSimplePath(P3)
N := findSimplePathsWithValue(i_1, a)
for all n ∈ N do
    P := P ∪ getValuesAfterPosition(i_2, n)
end for
```

**Algorithm 3** $children(a)$

**Require:** $a$ class name, $P1$ and $P3$ simple-paths
**Ensure:** $C$ set of children of $a$

```
C := ∅
v := lookupValue(a)
i_1 := computeSimplePath(P3)
i_2 := computeSimplePath(P1)
N := findSimplePathsWithValue(i_1, a)
for all n ∈ N do
    C := C ∪ getValuesBeforePosition(i_2, n)
end for
```

**Algorithm 4** $ancestors(a)$

**Require:** $a$ class name
**Ensure:** $A$ set of ancestors of $a$

```
A := ∅
P := parents(a)
A := P
for all p ∈ P do
    A := A ∪ ancestors(p)
end for
```

**Algorithm 5** $descendants(a)$

**Require:** $a$ class name
**Ensure:** $D$ set of descendants of $a$

```
D := ∅
C := children(a)
D := C
for all c ∈ C do
    D := D ∪ descendants(c)
end for
```

**Algorithm 6** $leaves(a)$

**Require:** $a$ class name
**Ensure:** $L$ set of leaves of $a$

```
L := ∅
C := children(a)
if C == ∅ then
    L := L ∪ {a}
else
    for all c ∈ C do
        L := L ∪ leaves(c)
    end for
end if
```

**Algorithm 7** $nca(a_1, \ldots, a_n)$

**Require:** $a_1$, …, $a_n$ class names
**Ensure:** $NCA$ the nearest common ancestor of $a_1, \ldots, a_n$

```
NCA := ∅
for k = 1 to n do
    CA := ancestors(a_k)
    NCA := NCA ∩ CA
end for
if NCA == ∅ then
    NCA := Thing
else
    while |NCA| > 1 do
        for j = 1 to |NCA| do
            for k = j + 1 to |NCA| do
                if NCA_j ∈ parents(NCA_k) then
                    NCA := NCA/{NCA_j}
                    j := j − 1
                    k := |NCA|
                else if NCA_k ∈ parents(NCA_j) then
                    NCA := NCA/{NCA_k}
                    k := k − 1
                end if
            end for
        end for
    end while
end if
```

**Algorithm 8** $properties(a)$

**Require:** $a$ class name, $P2$ and $P4$ simple-paths
**Ensure:** $P$ list of properties having $a$ as domain

```
P := ∅
i_1 := computeSimplePath(P4)
i_2 := computeSimplePath(P2)
A := ancestors(a) ∪ a
for all a ∈ A do
    N := findSimplePathsWithValue(i_1, a)
    for all n ∈ N do
        P := P ∪ getValueBeforePosition(i_2, n)
    end for
end for
```

**Algorithm 9** $keyword\text{-}based\_search(A_1, \ldots, A_n)$

**Require:** $a_1, \ldots, a_n$ names to search, $P1$ simple-path
**Ensure:** $C$ set of classes syntactically similar to $a_1, \ldots, a_n$

```
C := ∅
i := computeSimplePath(P1)
for k = 1 to n do
    V := lookupValueLike(a_k)
    for all v ∈ V do
        if v! = null then
            n := findSimplePathsWithValue(i, v)
            if n! = ∅ then
                C := C ∪ {lookupCode(v)}
            end if
        else
            n := findSimplePathsWithValueLike(i, v)
            if n! = ∅ then
                C := C ∪ {v}
            end if
        end if
    end for
end for
```

**Algorithm 10** $entity\text{-}based\_search(a)$

**Require:** $a$ class name, $P5$, $P6$ and $P7$ simple-paths
**Ensure:** $Ins$ list of individuals instance of $A$

```
Ins := ∅
i_1 := computeSimplePath(P6)
i_2 := computeSimplePath(P5)
i_3 := computeSimplePath(P7)
C := descendants(a) ∪ {a}
for all c ∈ C do
    N := findSimplePathsWithValue(i_1, c)
    M := findSimplePathsWithValue(i_3, c)
    for all n ∈ N do
        Ins := Ins ∪ getValuesBeforePosition(i_2, n)
    end for
    for all m ∈ M do
        Ins := Ins ∪ getValuesBeforePosition(i_2, m)
    end for
end for
```

**Algorithm 11** $entity\text{-}based\_search(a_1, \ldots, a_n)$

**Require:** $a_1, \ldots, a_n$ classes names, $P5$ and $P7$ simple-paths
**Ensure:** $Ins$ list of individuals instance of the intersection of $a_1, \ldots, a_n$

```
Ins := ∅
i_1 := computeSimplePath(P7)
i_2 := computeSimplePath(P5)
for 1 := 1 to n do
    C_i := descendants(a_i) ∪ {a_i}
    for all c ∈ C_i do
        N := findSimplePathsWithValue(i_1, c)
        for all n ∈ N do
            Ins_i := Ins_i ∪ getValuesBeforePosition(i_2, n)
        end for
    end for
end for
Ins := Ins_1 ∩ . . . ∩ Ins_n
```

| Name | Input | Output | Complexity |
|---|---|---|---|
| $lookupTag(t)$ | tag name $t$ | interval or *null* | $O(\|\mathcal{H}_T\|)$ |
| $lookupValue(v)$ | string value $v$ | code of $v$ or $v$ itself | $O(\|\mathcal{H}_V\|)$ |
| $lookupValueLike(v)$ | string value $v$ | (possibly empty) set of codes of strings containing $v$ | $O(\|\mathcal{H}_V\|)$ |
| $lookupCode(c)$ | code $c$ between 0 and 253 | string at position $c$ in $\mathcal{H}_V$ or *null* | $O(\|\mathcal{H}_V\|)$ |
| $computeSimplePath(P)$ | simple-path $P$ | interval or *null* | $O(\|P\| \times \|\mathcal{H}_T\|)$ |
| $getNextToken(n)$ | position $n$ | next token $t$ | $O(1)$ |
| $getPreviousToken(n)$ | position $n$ | previous token $t$ | $O(1)$ |
| $isStartToken(t)$ | token $t$ | *true* or *false* | $O(1)$ |
| $getNextValue(n)$ | position $n$ | (possibly encoded) string | $O(1)$ |
| $findSimplePathsWithValue(i, v)$ | interval $i$ of simple-path, string value $v$ | (possibly empty) set of occurrences, as positions from start of $\mathcal{B}$ | $O(\|\mathcal{B}\| + \|\mathcal{H}_V\|)$ |
| $findSimplePathsWithValueLike(i, v)$ | interval $i$ of simple-path, string value $v$ | (possibly empty) set of occurrences, as positions from start of $\mathcal{B}$ | $O(\|\mathcal{B}\|)$ |
| $getValuesAfterPosition(i, n)$ | interval $i$, position $n$ | (possibly empty) set of strings | $O(\|\mathcal{B}\| + \|\mathcal{H}_V\|)$ |
| $getValuesBeforePosition(i, n)$ | interval $i$, position $n$ | (possibly empty) set of strings | $O(\|\mathcal{B}\| + \|\mathcal{H}_V\|)$ |

TABLE I
ACCESS PRIMITIVES FOR A COX COMPRESSED DOCUMENT

space), a possible set of executable queries is the following:
– $keyword\text{-}based\_search(aid\,kit, sensor, weather)$: Algorithm 9 suggests *First_aid_kit, Sensor* and *Weather_Sensor* classes. It is useful to point out that a keyword-based search gives support to select suitable ontology classes, but it is not a necessary pre-condition for other types of ABox queries.
– $entity\text{-}based\_search(First\_aid\_kit, Weather\_Sensor)$: Algorithm 11 returns all the individuals that are instance of the intersection of the input concepts. The individual in Figure 1 is returned because $Thermometer$ class is subsumed by $Weather\_Sensor$ class and the remaining class is in the individual definition. So, the problem of instance retrieval can be solved.
– $owl : Class//@rdf : about/First\_aid\_kit \wedge (owl : Class//((owl : restriction//measures) \leftrightarrow (owl : restriction//owl : someValuesFrom))$: a generic path-based query that retrieves the instances of a general concept expression. It is expressed according to definitions in Section IV-A and on the RDF/XML syntax exemplified in Figure 1. It is executed using the *computeSimplePath* COX access primitive. Now, in order to retrieve all the individuals instance of the concept expression (*i.e.*, equivalent resources and more specific ones), the query has to be rewritten according to recursive application of normalization rules (see concept-normal form rules in [15] for more details). Also in this case, the individual in Figure 1 is returned because it is instance of intersection of $First\_aid\_kit$ class and unqualified existential restriction on $measures$ property. In the devised query framework, path-based search on semantically annotated instances solves the problem of query answering on the ABox.

## VI. CONCLUSION

In the context of semantic-based data management for pervasive computing, a framework has been presented for querying knowledge bases expressed in OWL, serialized in RDF/XML and compressed with a homomorphic algorithm. The provided query engine formalization, based on a set of data access primitives, has demonstrated the feasibility of the approach, the absence of algorithmic complexity issues and an acceptable theoretical scalability. Implementation of the framework is ongoing. It will allow extensive experimental analysis, which is needed to compare performance of our query processing strategy with other existing ones as well as to evaluate possible optimizations. Future developments include: (i) a full-featured query and reasoning engine for compressed KBs; (ii) integration of queries on compressed KBs in a protocol for semantic resource discovery in pervasive environments.

## REFERENCES


[1] F. Scioscia and M. Ruta, "Building a Semantic Web of Things: issues and perspectives in information compression," in *SWIM'09*. IEEE Computer Society, 2009, pp. 589–594.
[2] R. De Virgilio, E. Di Sciascio, M. Ruta, F. Scioscia, and R. Torlone, "Semantic-based RFID Data Management," in *Unique Radio Innovation for the 21st Century: Building Scalable and Global RFID Networks*. Springer, 2010.
[3] T. Di Noia, E. Di Sciascio, F. M. Donini, M. Ruta, F. Scioscia, and E. Tinelli, "Semantic-based Bluetooth-RFID interaction for advanced resource discovery in pervasive contexts," *IJSWIS*, vol. 4, no. 1, pp. 50–74, 2008.
[4] M. Ruta, G. Zacheo, A. L. Grieco, T. Di Noia, G. Boggia, E. Tinelli, P. Camarda, and E. Di Sciascio, "Semantic-based Resource Discovery, Composition and Substitution in IEEE 802.11 Mobile Ad Hoc Networks," *WiNet*, vol. 16, no. 5, pp. 1223–1251, 2010.
[5] J. Vazquez and D. López-de Ipiña, "mRDP: An HTTP-based lightweight semantic discovery protocol," *Computer Networks*, vol. 51, no. 16, pp. 4529–4542, 2007.
[6] J. Min, M. Park, and C. Chung, "A compressor for effective archiving, retrieval, and updating of XML documents," *TOIT '06*, vol. 6, no. 3, pp. 223–258, 2006.
[7] S. Sakr, "XML compression techniques: A survey and comparison," *JCSS*, vol. 75, no. 5, pp. 303–322, 2009.
[8] P. Tolani and J. Haritsa, "XGRIND: A Query-friendly XML Compressor," in *ICDE '02*. IEEE, 2002, pp. 225–234.
[9] P. Skibiski and J. Swacha, "Combining Efficient XML Compression with Query Processing," in *ADBIS*. Springer, 2007, vol. 4690, pp. 330–342.
[10] V. Christophides, D. Plexousakis, M. Scholl, and S. Tourtounis, "On labeling schemes for the Semantic Web," in *WWW '03*. ACM, 2003, pp. 544–555.
[11] S. Sakr and G. Al-Naymat, "Relational processing of RDF queries: a survey," *SIGMOD Rec.*, vol. 38, pp. 23–28, June 2010.
[12] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix Bit loaded: a scalable lightweight join query processor for RDF data," in *WWW '10*. ACM, 2010, pp. 41–50.
[13] R. Delbru, N. Toupikov, M. Catasta, and G. Tummarello, "A Node Indexing Scheme for Web Entity Retrieval," in *The Semantic Web: Research and Applications*, 2010, vol. 6089, pp. 240–256.
[14] Y. Zhang, N. Zhang, J. Tang, J. Rao, and W. Tang, "Mquery: Fast graph query via semantic indexing for mobile context," in *WI-IAT '10*. IEEE Computer Society, 2010, pp. 508–515.
[15] F. Baader, D. Calvanese, D. Mc Guinness, D. Nardi, and P. Patel-Schneider, Eds., *The Description Logic Handbook*. Cambridge University Press, 2003.