Searching Semantic Web Objects Based on Class Hierarchies

Gong Cheng
gcheng@seu.edu.cn

Weiyi Ge
wyge@seu.edu.cn

Honghan Wu
hhwu@seu.edu.cn

Yuzhong Qu
yzqu@seu.edu.cn

Institute of Web Science, School of Computer Science and Engineering
Southeast University, Nanjing 210096, P.R. China

ABSTRACT

On the Semantic Web, objects can be identified by URIs and described by using RDF. With more and more RDF data published, object search on the Semantic Web is in demand for browsing, reuse, and integration. We developed the Falcons system, a keyword-based search engine for the Semantic Web. In this paper, we present the object search service provided by Falcons. For building an inverted index from query terms to Semantic Web objects (SW objects), we present a method to construct comprehensive textual description of SW objects. Furthermore, SW objects are also indexed from their classes and ancestor classes to support the class-based query restriction. Especially, class subsumption reasoning on multiple vocabularies on the Semantic Web is performed, and a class recommendation technique is proposed to enable hierarchically navigating classes. In addition, a summarization method for SW objects is devised to enable users to browse the summaries of objects.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Design, Experimentation

Keywords

Class Hierarchy, Indexing, Navigation, Search Engine, Summarization

1. INTRODUCTION

A Web of data indicates a large amount of interlinked data. Linked data encourages to use HTTP URIs as names for things, and link them with other URIs so that data consumers can discover more information. People may create new URIs for all the things to be described, despite the possible high maintenance cost. Such style of writing can produce a large amount of data, but the data may be not interlinked. Actually, the Linking Open Data community[1] have put much effect into setting links between data items from different data sources. However, it is too difficult to interlink all the existing data after they born. Instead, people could reuse existing URIs to identify the things to be described in their RDF documents. For example, we can find the URIs that identify geographical names on GeoNames[2], and can find the URIs that identify authors and papers on DBLP Berlin[3]. But, apparently, not everyone knows all these open data sources, and not everyone would like to switch frequently between various data sources to search for URIs when writing data.

Nowadays, on the Web of documents, search engines help people find documents quickly, and promote the development of the Web itself. Analogously, on the Web of data, we believe that a well-designed search engine can also help people find Semantic Web entities (URIs that are used on the Semantic Web to denote concepts and objects) for the information needs as well as the purpose of reusing. Therefore, we developed Falcons[4], a keyword-based search engine for Semantic Web entities. In this paper, we present the object search service in Falcons. Different from other keyword-based Semantic Web search engines [6, 7, 11, 14], we expand the textual description of SW objects from their associated literals to also include the textual description of their neighboring SW objects in RDF graphs, so that desired SW objects will be less likely to be missing for keyword queries, and we also develop a weighting scheme for preserving the precision. In order to help users quickly find SW objects, we index the classes of SW objects and provide a user-friendly navigation hierarchy of classes for users to refine the search results. In addition, for each SW object, a set of RDF statements about it is extracted from various data sources and organized into a summary to show its meaning (denotation) for users to decide whether it is the one needed.

The remainder of the paper is structured as follows. Section 2 demonstrates the approach with the Falcons system and presents its architecture. Section 3 introduces a method to expand the textual description of SW objects for building the inverted index. Section 4 describes a way to enable navigating class hierarchies for query restriction. Problems and

[1] `http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData`.

LDOW2008, April 22, 2008, Beijing, China.

[2] `http://www.geonames.org/`.

[3] `http://www4.wiwiss.fu-berlin.de/dblp/`.

[4] `http://iws.seu.edu.cn/services/falcons/`.

Figure 1: A screenshot of Falcons Object Search.

challenges in integrating class hierarchies on the Semantic Web are discussed in Section 5. A preliminary implementation of SW object summarization is described in Section 6. Related work is presented in Section 7. Conclusion and future work are given in Section 8.

2. SYSTEM OVERVIEW

2.1 Demo

In Falcons Object Search, after being fed by keywords, say "Tim Berners-Lee", Falcons will match these terms with the textual description of SW objects and serve a list of ranked results. Each SW object is associated with its types and labels to help the user quickly understand its denotation. If such information is insufficient, users can click on the reading glass beside each SW object to obtain a more detailed summary. We also present the number of RDF documents where it is used, to show its popularity, linked to the page listing these documents. Besides, at the top of the search results page, several tags are dynamically generated and recommended, indicating the types of SW objects that the user is probably searching for. After clicking on a tag, say "Person" here, the results will be refined to only include the SW objects of the type "Person" (including `foaf:Person`, `swrc:Person`, etc.). Meanwhile, more specific types, e.g., "Engineer" and "Scientist", will be recommended to the user, as depicted in Fig. 1. The user can proceed to more specific types to refine the search results further, or return to more general ones using the track bar.

2.2 Architecture

Fig. 2 depicts the architecture of the Falcons system. In Falcons, an RDF crawler is set up to download RDF documents. General keywords are enumerated and combined as queries, sent to Google[5] and Swoogle[6] to obtain seeds. The crawler is also customized to download RDF documents from DBpedia[7], Hannover DBLP Bibliography[8], Ping the Semantic

[5]`http://www.google.com/`.
[6]`http://swoogle.umbc.edu/`.
[7]`http://dbpedia.org/`.
[8]`http://dblp.l3s.de/d2r/`.

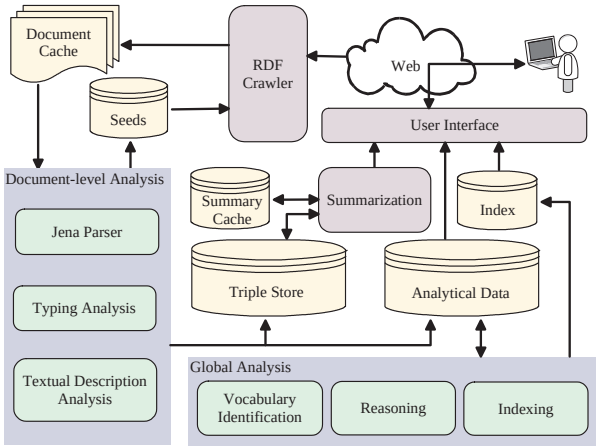

Figure 2: The architecture of Falcons Object Search.

Web.com[9], SchemaWeb[10], etc. At the time of writing, more than 7 million well-formed RDF documents, containing 250 million RDF statements, have been successfully downloaded.

RDF documents are parsed by Jena [11]. RDF statements and the intermediate results of document-level analysis are stored in MySQL[12] database. All the URIs discovered in the documents are sent to the seeds database for crawling. The global analysis distinguishes between concepts and objects, identifies valid vocabularies, and performs the indexing periodically. At the time of writing, about 30 million SW objects have been indexed. Finally, the summarization module computes and caches summaries of SW objects.

3. MAPPING TERMS TO SW OBJECTS

On the Web of documents, keyword-based search engines extract terms from documents and map query terms to documents based on an inverted index. On the Web of data, SW objects are identified by URIs, from which only limited useful terms (maybe just local names) can be extracted. So a widely used method is to use both local names and associated literals of SW objects to form their textual description and then build the inverted index.

Actually, in RDF, semantic links can be established from SW objects to either SW objects (by using object properties) or literals (by using datatype properties). They are just two styles of describing SW objects, so only indexing the terms in the literals may cause that some potentially matched SW objects are missing for some queries, considering that query terms may indicate anything. For example, Fig. 3 depicts that the SW object `wiki:WWW2008` identifies the WWW2008 conference held in Beijing. If just the terms extracted from the literals are indexed, namely "Conference", "17th International World Wide Web Conference", and "WWW2008" here, this SW object will not be retrieved when the user states "conference" and "Beijing" to indicate the conferences held in Beijing because "Beijing" is not indexed. Such ob-

[9]`http://pingthesemanticweb.com/`.
[10]`http://www.schemaweb.info/`.
[11]`http://jena.sourceforge.net/`.
[12]`http://www.mysql.com/`.

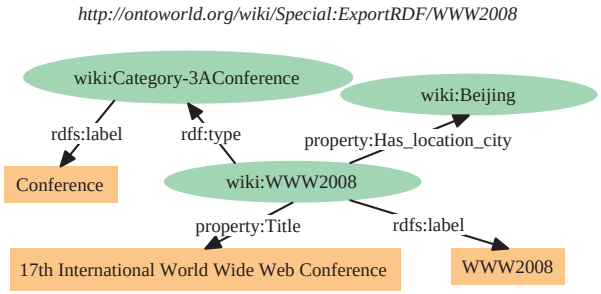

Figure 3: A fragment of the RDF statements about wiki:WWW2008.

servation motivates us to expand the textual description of SW objects from their associated literals to also include the textual description of their neighboring SW objects in RDF graphs.

3.1 Expanding Textual Description of SW Objects

In graph theory, the neighbors of a vertex is defined to be the vertices adjacent to it (excluding itself), i.e., an edge exists between a vertex and its neighbor. However, such definition is not suitable for RDF graphs owing to the use of blank nodes. In RDF, blank nodes indicate the existence of things without being assigned global identifiers, and we cannot extract any terms from blank nodes. Actually, blank nodes are created for connecting other SW entities or literals. So in a traversing view, starting from some SW object, we traverse links to collect vertices as its neighbors and stop until reaching URIs or literals (but not blank nodes).

To formalize, we use a notion of *RDF sentence* [17]. In brief, an RDF sentence is a set of RDF statements that contain common blank nodes. Fig. 4 gives an example. For an RDF sentence s, let $\mathrm{Subj}(s)$ and $\mathrm{Obj}(s)$ be the set of URIs placed at the subject, and the set of URIs or literals placed at the object of any RDF statement in s, respectively. For example, for the RDF sentence s in Fig. 4, $\mathrm{Subj}(s) = \{$`http://kmi.open.ac.uk/people/tom/`$\}$ and $\mathrm{Obj}(s) = \{$"`UK Centre for Materials Education`", `foaf:Project`, `http://www.materials.ac.uk/`$\}$. An equivalent definition of RDF sentence is the *Minimum Self-Contained Graph* [16]. Following the theorems in [16], an RDF graph g can be decomposed into a unique set of RDF sentences, denoted by $\mathrm{Sent}(g)$. Then, in g, the *neighbors* of a SW object o, denoted by $\mathrm{Neib}(o, g)$, can be formulated as follows:

$$\mathrm{Neib}(o, g) = \bigcup_{\substack{s \in \mathrm{Sent}(g) \\ o \in \mathrm{Subj}(s)}} \mathrm{Obj}(s). \tag{1}$$

Readers will notice that we collect neighbors by traversing only forward links. Actually, on the Web of data, forward and backward links should be treated equivalently. But we find that only traversing forward links works well enough in practice, while traversing backward links may bring too many neighbors for "popular" SW objects, which is difficult to process.

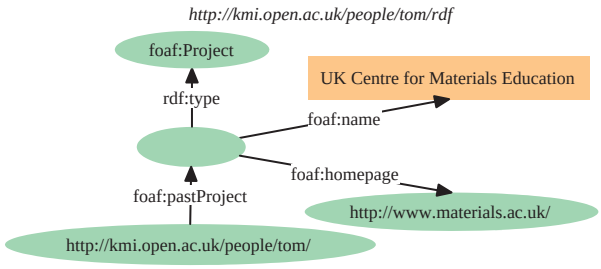

Figure 4: An RDF sentence.

We use the well-known Vector Space Model (VSM) to represent terms, that is, either the textual description of a SW object or the textual description of a keyword query is represented as a word vector in the word vector space.

We first consider a single RDF document, and we use g to denote the RDF graph serialized by the document. For a URI r, let $\mathrm{LN}(r)$ denote the word vector representing its local name, and let $\mathrm{Lab}(r, g)$ and $\mathrm{Comm}(r, g)$ denote the word vectors representing its labels (values of `rdfs:label`) and comments (values of `rdfs:comment`) in g, respectively. For a literal r, let $\mathrm{LexForm}(r)$ denote the word vector representing its lexical form. Then, for a URI or literal r, we define $\mathrm{LocText}(r, g)$ as the word vector representing its local textual description, or "name", as follows:

$$\mathrm{LocText}(r, g) = \begin{cases} \mathrm{Lab}(r, g), & \text{for a URI } r,\ \mathrm{Lab}(r, g) \neq \mathbf{0} \\ \mathrm{LN}(r), & \text{for a URI } r,\ \mathrm{Lab}(r, g) = \mathbf{0} \\ \mathrm{LexForm}(r), & \text{for a literal } r. \end{cases} \tag{2}$$

Then, as mentioned, for each SW object o, we index not only its basic textual description such as its local name and labels but also the textual description from its neighbors in g:

$$\begin{aligned} \mathrm{Text}(o, g) =& \alpha \cdot \mathrm{LN}(o) + \beta \cdot \mathrm{Lab}(o, g) + \gamma \cdot \mathrm{Comm}(o, g) \\ &+ \sum_{r \in \mathrm{Neib}(o, g)} \mathrm{LocText}(r, g), \end{aligned} \tag{3}$$

where α, β, and γ are the weighting coefficients to be tuned, and are set to 10, 5, and 2 in Falcons, respectively.

Consequently, for `wiki:WWW2008` in Fig. 3, besides the terms in its associated literals, the terms "Beijing" (from `wiki:Beijing`) and "Conference" (from `wiki:Category-3AConference`) are also indexed to it so that users can find it with any of these terms. Apparently, compared to traditional methods, more SW objects are indexed from each term owing to the introduction of terms from neighboring SW objects. So we use the weighting scheme in (3) to ensure that well matched SW objects, e.g., whose labels match the query terms, can be ranked higher.

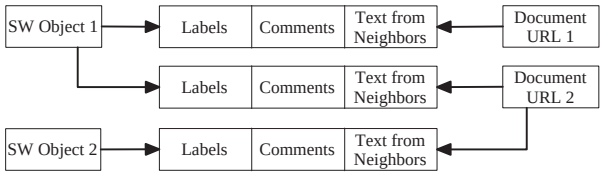

Figure 5: Recording the textual description of SW objects from different documents on database.

Finally, notice that on the Web of data, a SW object may be described by many documents. To enable cross-document search, for each SW object o, we aggregate its textual description from all these documents:

$$\text{Text}(o) = \sum_{g} \text{Text}(o, g). \quad (4)$$

3.2 Implementation

There are two major steps in computing the textual description of SW objects. Firstly, it requires an algorithm to find all the neighbors for each SW object. The key point is to decompose an RDF graph into a set of RDF sentences. To achieve this, we transform an RDF graph into an undirected statement graph, where vertices represent RDF statements in the original RDF graph, and an edge exists between two statements iff they contain common blank nodes. Then each connected component in the statement graph corresponds to an RDF sentence. Alternatively, we can also use depth-first search to traverse the original RDF graph to find all the neighbors for each SW object.

Secondly, when computing the global textual description of each SW object according to (4), it is inefficient to parse all the documents on the fly. Instead, we parse each document just for once. As depicted in Fig. 5, for each document, we extract for each SW object (described by this document) its labels, comments and the terms from its neighbors, and record them on database. Indexes are created on both SW objects and document URLs. The index on SW objects is used to collect all the terms from different documents for each SW object; the index on document URLs is used for document updating. Finally, we compute the global textual description of SW objects and update the inverted index periodically. The inverted index is implemented based on Apache Lucene[13].

4. NAVIGATING CLASS HIERARCHIES FOR QUERY RESTRICTION

We use information retrieval (IR) techniques to implement the index and provide search. However, such plain term-matching method often returns a large amount of results, most of which are not interesting to users. It is because users lack ways to precisely restrict the queries. The same problem is encountered in Web document search. To help users find more relevant information in a shorter time, Web documents can be classified into predefined categories so that users can restrict the queries and find more accurate documents by specifying the desired category [5].

[13]`http://lucene.apache.org/`.

On the Semantic Web, we can also classify SW objects into predefined categories such as the Open Directory Project[14]. But actually, most SW objects were born and classified precisely into one or more classes by using `rdf:type`. We can utilize such information to provide interfaces for users to restrict the queries. Some Semantic Web search engines such as SWSE [11] and an old version of our Falcons Object Search have implemented this method. Nevertheless, after two-month running, we have learnt two major lessons. Firstly, indexing only explicitly stated types is insufficient. For example, in Fig. 3, `wiki:WWW2008` is explicitly stated to be an instance of `wiki:Category-3AConference` (conference); we also know that `wiki:Category-3AConference` is a subclass of `wiki:Category-3AResearch_event` (research event)[15]. However, when `wiki:Category-3AResearch_event` is selected to restrict the query, `wiki:WWW2008` will not be returned because no reasoning is performed. Secondly, in many cases, a lot of classes are recommended to users as candidate restrictions but these classes present low diversify, e.g., `wiki:Category-3AResearch_event` and its subclass `wiki:Category-3AConference` may be recommended simultaneously.

To solve these problems, we perform reasoning to discover implicitly stated classes for SW objects. Besides, it is common to manage complexity by using hierarchy [4], so we provide users with a navigation hierarchy of classes instead of a simple list to restrict the queries. Techniques that make possible such capabilities will be presented in detail in the following subsections.

4.1 Vocabularies on the Semantic Web

Before we perform reasoning on the collected class hierarchies from the Semantic Web, we must solve several fundamental problems, e.g., which URIs identify classes and which RDF statements (axioms) will be accepted by the reasoning engine, because, on the Semantic Web, anyone can say anything.

Here we formulate a *vocabulary* v as a quadruple $< uri, C, P, D >$, where uri is the URI identifying this vocabulary, C is the set of classes in this vocabulary, P is the set of properties in this vocabulary, and D is a set of RDF documents that "define" this vocabulary. Details about these components are given in the following.

Most vocabularies on the Semantic Web use a "hash namespace" or a "slash namespace" [12]. For example, the SKOS Core Vocabulary uses a "hash namespace" `http://www.w3.org/2004/02/skos/core`. The URIs of the classes and properties in such vocabularies are constructed by appending a hash character ('`#`') and a local name to the vocabulary URI, e.g., `http://www.w3.org/2004/02/skos/core#Concept`. The FOAF vocabulary uses a "slash namespace". In this case, the vocabulary URI ends with a forward slash character ('`/`'), and the URIs of the classes and properties in such vocabularies are constructed by directly appending a local name to the vocabulary URI, e.g., `http://xmlns.com/foaf/0.1/Person`. We use these rules to find out candidate

[14]`http://www.dmoz.org/`.

[15]See `http://ontoworld.org/wiki/Special:ExportRDF/Category:Conference`.

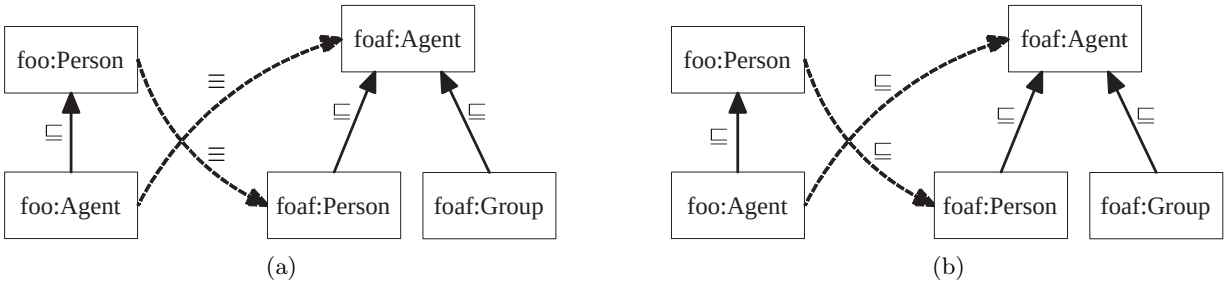

Figure 6: Transforming cross-vocabulary (a) class equivalence axioms into (b) class inclusion axioms.

classes (properties) for each vocabulary. Next, some classes (properties) will be cleaned from corresponding vocabularies. For example, we find that a class `foaf:#Person` (NOT `foaf:Person`) is used in many RDF documents (actually it is a mistake in writing). Apparently this class is not in the current FOAF vocabulary and should be cleaned out. Theoretically, people or organizations who declare URIs should own or be delegated to own these URIs in the real world. So for simplicity, we accept that a class (property) belongs to its corresponding vocabulary if we find that the URI of the class (property) is used to identify a class (property) in (at least) one RDF document satisfying that the host component [3] of the document URL coincides with the host component of the vocabulary's URI. By combining this rule and the previous string manipulation technique, all the classes (properties) of a vocabulary can be determined.

There are at least two ways for developers to define a vocabulary. A convenient way for small vocabularies such as FOAF is to encode all the axioms about a vocabulary in one RDF document and serve it when the vocabulary URI is dereferenced[16]. Alternatively, for large vocabularies such as the YAGO Classification (used by DBpedia), only a subset of axioms about a class is served when the URI of the class is dereferenced. So for a vocabulary v, $v.D$ includes a possible RDF document obtained by dereferencing $v.uri$, and at most $|v.C|$ RDF documents obtained by dereferencing the URIs of the classes in $v.C$[17].

4.2 Class Subsumption Reasoning on Multiple Vocabularies

For simplicity, only the class inclusion axioms on simple classes (identified by URIs) are considered in this paper. However, it is not appropriate to perform reasoning on all such axioms decoded from all the discovered documents because, e.g., one can easily mess up the system by encoding `rdfs:Resource` $\sqsubseteq$ `foaf:Person` in some document.

Therefore, as a preliminary step, we perform reasoning on a conservative set of axioms. Formally, for each vocabulary v, only those axioms decoded from $v.D$ are allowed to constrain the meaning of the classes in $v.C$. It is inspired by the ontology modularization work in [9]. In other words, each vocabulary is allowed to reuse the classes from other vocabularies but cannot further constrain their meaning. As an example, for the axiom `rdfs:Resource` $\sqsubseteq$ `foaf:Person`, it will be accepted by the reasoning engine iff it comes from some document in RDFS.D.

[16]In this paper, the 'Accept' field in the header of the HTTP request is set to 'application/rdf+xml' when we say "dereference".

[17]Properties are not considered in class-based query restriction.

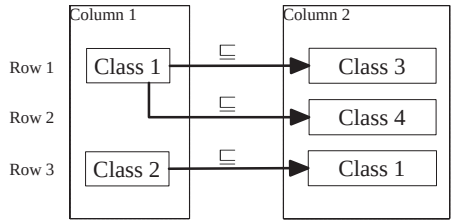

Figure 7: Recording the class inclusion axioms on database.

Specifically, class equivalence axioms are widely used on the Semantic Web. A synthetic vocabulary FOO is depicted in Fig. 6(a), in which two class equivalence axioms are used to characterize the classes in the vocabulary. If these axioms are simply accepted by the reasoning engine, it can be inferred that `foaf:Agent` and `foaf:Person` are equivalent, which is unexpected. Actually, these axioms violate the previous principle because they constrain the meaning of some classes (`foaf:Agent` and `foaf:Person`) out of the FOO vocabulary. However, in order to preserve the vocabulary authors' original intention, we do not simply reject these axioms, but instead, we transform them into class inclusion axioms, as depicted in Fig. 6(b). Then the meaning of the classes in the FOO vocabulary is more or less constrained while the meaning of the classes in the FOAF vocabulary is not further constrained.

By applying these principles, we can determine which axioms will be accepted by the reasoning engine, and then we can perform reasoning. Actually, the reasoning can be performed either at search time or at indexing time. In the former case, when one class is selected to restrict the query, all its subclasses can be inferred on the fly and used to expand the restriction. However, it will definitely reduce the search efficiency as well as the user experience. So we choose the latter one.

Axioms are recorded on a two-column table R in database, as depicted in Fig. 7. Three basic functions, LoadSupsOf(c, R),

Table 1: The algorithm to compute the transitive closure of the class subsumption relation

```
1.  procedure(R)
2.    Set C := LoadAllClasses(R);
3.    while C ≠ ∅
4.      choose c from C;
5.      Set axioms := ∅;
6.      Set traversed := ∅;
7.      Set untraversed := {c};
8.      while untraversed ≠ ∅
9.        choose s from untraversed;
10.       remove s from untraversed;
11.       remove s from C;
12.       if s ∉ traversed
13.         add s to traversed;
14.         for each s' in LoadSupsOf(s, R)
15.           add s ⊑ s' to axioms;
16.           if s' ∉ untraversed AND s' ∉ traversed
17.             add s' to untraversed;
18.           end if;
19.         end for;
20.       end if;
21.     end while;
22.     Set axioms+ := TransitiveClosure(axioms);
23.     for each a in axioms+
24.       Write(a, R);
25.     end for;
26.   end while;
27. end procedure;
```

LoadAllClasses(R), and Write(a, R) are defined on R: LoadSupsOf(c, R) returns all the superclasses of the class c in the current R; LoadAllClasses(R) returns all the distinct classes in column 1 of R; Write(a, R) inserts a class inclusion axiom a into R if a does not exist in R. These functions can be implemented using SQL statements easily. Based on these functions and TransitiveClosure, the transitive reasoner provided by Jena, we implement a simple algorithm to compute all the superclasses of each class (including explicitly stated and inferred) and write them back to R. In other words, we will compute the transitive closure of the class subsumption relation.

The algorithm is presented in Table 1. In brief, for each class, all its superclasses as well as the class inclusion axioms are collected in a breadth-first-search way (lines 4 to 21). Next, the transitive reasoner in Jena is called to compute the transitive closure of this set of collected axioms, and then they are written back to R (lines 22 to 25).

The algorithm can be significantly optimized. Firstly, it can be easily converted into a parallel version. Secondly, if the class chosen at line 4 has never been instantiated, it is not necessary to continue this run of the loop because, either all its superclasses will be computed when the algorithm processes one of its subclass (been instantiated), or it is not necessary at all to compute them since they will never be used in the next indexing step.

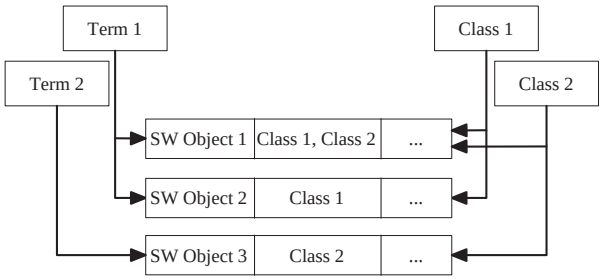

Figure 8: Indexing the types of SW objects.

Readers will note that the proposed method cannot deal with vocabulary updating. It is too complicated to support updating because we will have to keep track of where each axiom originates from. Nevertheless, we find that, based on more than 4,000 vocabularies our system has discovered, it takes no more than 3 hours to accomplish the computation of the transitive closure (including parsing documents), which is tolerable compared to the update period of the system.

4.3 Indexing Classes of SW Objects

In the system, there are two ways to use the transitive closure of the class subsumption relation. When one class is selected to restrict the query, we can look up all its subclasses in R to expand the restriction. In this way, each of these subclasses is represented as a clause in the constructed boolean query. However, the search efficiency will greatly decrease and the search may even fail to proceed when the query contains too many clauses. Thus, alternatively, for each SW object, we directly index all its classes (including the inferred ones). In this way, more disk space is required to store the index but the search efficiency is not affected.

The indexing scheme is depicted in Fig. 8. On one hand, based on the inverted index (from terms to SW objects), each term in the query is mapped to a set of SW objects. On the other hand, based on the index from classes to SW objects, each class is mapped to its instances, which is also a set of SW objects. Then for a keyword query with class restrictions, the intersection of such sets are served as the results and returned to the user. All these capabilities can be easily implemented based on Apache Lucene.

The SW objects in the results are ranked by a combination of their relevance to the query and their popularity. The relevance score is calculated by using the TF/IDF technique, which is natively supported by Apache Lucene. The popularity score is calculated based on the number of RDF documents that the SW object is used in.

4.4 Navigating Class Hierarchies

The class hierarchy in this case is different from the category hierarchy used by many E-Commerce sites. On E-Commerce sites, the category hierarchy is single, carefully designed, and relatively small-scale (generally including hundreds of categories). But the class hierarchy on the Semantic Web comprises a large amount of class hierarchies from different vocabularies of various qualities. At the time of writing, Falcons has discovered more than 2 million classes, and more than 200 thousand of them (or their subclasses) have been

instantiated. Owing to the large scale, many general indexing methods cannot be directly applied.

In Section 2, we have shown that we provide user-friendly tags, instead of the URIs of the classes, as candidate restrictions. Each tag may correspond to more than one classes. So we formulate a *user query* q as a two-tuple $< \mathbb{T}, \mathbb{C} >$, where $\mathbb{T}$ is a set of query terms and $\mathbb{C}$ is a set of classes for query restriction. A SW object is an *answer* to q iff its textual description contains all the terms in $q.\mathbb{T}$ and it is an instance of at least one class in $q.\mathbb{C}$. Initially, the user does not select any class restrictions, and $\mathbb{C}$ is implicitly set to {`rdfs:Resource`}. Then, navigating the hierarchy can be viewed as issuing a sequence of queries with the same set of query terms but different sets of class restrictions. Moving down (up) the hierarchy is to replace the class restrictions with more specific (general) ones.

So the search engine needs to determine which tags should be provided as candidate further restrictions at the next level. To achieve this, we devise a method composed of the following steps:

1. Find out all the answers to a given query $< \mathbb{T}, \mathbb{C} >$;
2. Collect the classes of the answers and rank them;
3. Select top K classes from the ranked list that satisfy the following constraints: (a) each selected class must be a strict subclass of some class in $\mathbb{C}$, and (b) the class subsumption relation does not hold between any pair of the selected classes;
4. Map the selected classes to tags and present them to the user.

We have presented the first step in Section 4.3. Next, for each SW object, all its classes are not only indexed but also stored in the index, as depicted in Fig. 8. So we can simply iterate over the answers to collect their classes. However, for some queries, there may be a large amount of answers so that iterating over all of them takes too much time and is intolerable in terms of user experience. To make trade off between the coverage and the efficiency, we only iterate over the first 1,000 answers. It is because, in most cases, users never browse that many answers. Actually some traditional Web search engines, such as Google, also just serve limited results.

Let $\tilde{\mathbb{C}}$ be the set of classes collected from the first 1,000 answers. We consider that, for each query, if one class covers more answers, it will be more likely to be recommended to the user as candidate further restrictions. When we construct $\tilde{\mathbb{C}}$, we also associate each class in $\tilde{\mathbb{C}}$ with the size of its instances in the answer set. Then, we rank the classes in $\tilde{\mathbb{C}}$ by the associated size in the descending order, and scan downwards the ranked list to accept at most K classes, denoted by $\tilde{\mathbb{C}}_K$. Each accepted class must satisfy two conditions. Firstly, it must be a strict subclass of some class in $\mathbb{C}$, i.e., $\tilde{c}$ will be accepted only if $\exists c \in \mathbb{C}, \tilde{c} \sqsubset c$. Secondly, it is required that all the accepted classes can provide good coverage. So one accepted class is not allowed to be a subclass of any other accepted classes, i.e., $\nexists \tilde{c_1}, \tilde{c_2} \in \tilde{\mathbb{C}}_K, \tilde{c_1} \sqsubseteq \tilde{c_2}$. When scanning a class $\tilde{c}$, if accepting it violates this condition owing to an accepted class $\tilde{c_0}$, we will either reject $\tilde{c}$ or replace $\tilde{c_0}$ with $\tilde{c}$: if $\tilde{c}$ is associated with a smaller size than $\tilde{c_0}$, $\tilde{c}$ will be rejected; if they are associated with the same size, we will accept the more specific one between them. All the subsumption checks are supported by the transitive closure of the class subsumption relation stored in R, as depicted in Fig. 7.

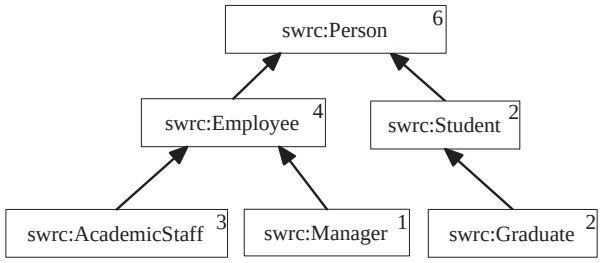

Figure 9: An example for navigation.

Finally, each class in $\tilde{\mathbb{C}}_K$ is mapped to a user-friendly tag. We just use its label, or, if cannot find any labels, use its local name. So it is possible that many classes are mapped to one tag.

As an example, Fig. 9 depicts a class hierarchy extracted from the SWRC vocabulary[18], and each class is associated with a number indicating the size of its instances in the answer set for a previous query. Suppose that the user has selected `swrc:Person` in the previous query. The following illustrates how to recommend tags at the next level. Here the ranked list of classes is: `swrc:Person`, `swrc:Employee`, `swrc:AcademicStaff`, `swrc:Student`/`swrc:Graduate`, and `swrc:Manager`, and let $K = 2$. Firstly, `swrc:Person` will not be accepted because it is not a strict subclass of the only class in the previous class restriction (itself). Next, its strict subclass `swrc:Employee` will be accepted, but then `swrc:AcademicStaff` will not be accepted because it is a subclass of `swrc:Employee` and is associated with a smaller size. Next, `swrc:Student` and `swrc:Graduate` are associated with the same size, but only `swrc:Graduate` will be accepted because it is more specific. So finally, the accepted classes will be `swrc:Employee` and `swrc:Graduate`, and the recommended tags at the next level will be "Employee" and "Graduate".

5. DISCUSSIONS ON INTEGRATING CLASS HIERARCHIES

There are many class hierarchies on the Semantic Web and some of them model overlapped domains. So it is necessary and also possible to integrate these hierarchies.

Integrating hierarchies is not a new topic. Recent ontology integration approaches and Web taxonomy integration approaches can be found in [1] and [18], respectively. Generally, there are at least two ways to integrate hierarchies. Firstly, we can insert [13] or map [2] classes from all the hierarchies into a selected master hierarchy. It is feasible to

[18] `http://swrc.ontoware.org/ontology`.

apply this approach to class hierarchies from the same domain. But in our case, vocabularies on the Semantic Web model various domains and it is difficult to find such a vocabulary that can cover all the domains. Actually if such a vocabulary exists, we even do not need any other ones. Secondly, we can integrate all the existing hierarchies into a new one [15]. However, this approach often requires a large amount of instances that simultaneously instantiate multiple classes from different hierarchies, but the current Semantic Web cannot provide.

In this paper, actually all the class hierarchies are integrated in a very preliminary way, i.e., only a conservative set of class inclusion axioms are used to connect different hierarchies, as described in Section 4.2. In this way, the "precision" of the integration can be guaranteed. However, inevitably, there are still a large amount of classes that can be but have not been merged, i.e., the "recall" is low.

One way to find such classes is to use ontology matching approaches [8]. But, two points restrict their use in this case. On one hand, ontology matching approaches are often designed to match a pair of ontologies. Then, we have to apply these approaches to all pairs of class hierarchies (numbered in the thousands). It will take much time to accomplish the task. On the other hand, most ontology matching approaches provide mappings as suggestions to human beings. So the resulting mappings are not stably reliable. In other words, the precision is not always 100%. And apparently, it is impractical to manually check all the resulting mappings, but even one error may mess up the whole hierarchy after the reasoning is performed.

Therefore, we do not use any matching approaches to provide third-party axioms. But in future work, we will still try to find appropriate ways to further integrate class hierarchies because it is necessary and useful in search engines.

6. SUMMARIZING SW OBJECTS

In the search results page, for each SW object, we present its types and labels for users to understand its denotation. But in some cases, types and labels may be ambiguous or even not available. Alternatively, users can directly dereference the URIs to view the content served by their owners. However, not all the URIs are dereferenceable, and the served Web pages or RDF data are limited. So some methods are still needed to help users understand what a SW object denotes and how to use it appropriately in RDF data.

To achieve this, for each SW object, we extract a set of RDF statements about it on the Semantic Web and organize them into a summary. A high-quality summary should be:

- *Informative*: the extracted statements should provide sufficient information about a SW object to help users understand.
- *Scalable*: the size of the extracted statements should be adaptable to serve different users.
- *Authoritative*: the extracted statements should derive from reliable data sources.

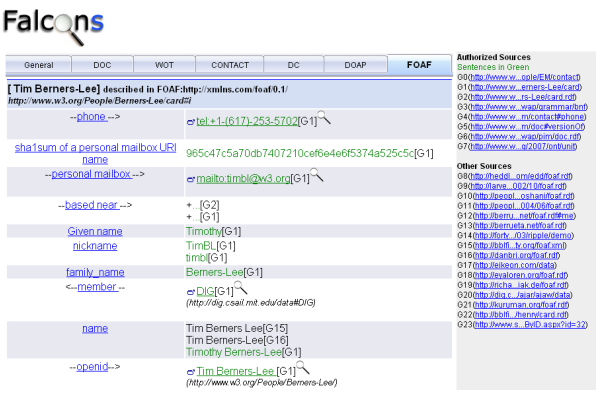

Figure 10: An example summary.

- *Comprehensive*: it is expected that the extracted statements describe many different aspects and derive from many different data sources.
- *Readable*: the extracted statements should be well organized and friendly presented.

So actually, the key point is to devise a method to evaluate and rank all the RDF statements for each SW object. However, we have not found any efficient algorithms that can work on hundreds of millions of dynamically updated statements.

At the time of writing, the SW object summarization in Falcons is still a preliminary work. Summaries are computed on the fly and then cached for future requests. For each SW object, all the RDF documents describing it are classified into three categories. The first category includes a possible RDF document served by dereferencing its URI. The RDF statements about the SW object in this document, often describing in a defining way, are more reliable than all the others. Besides, a SW object is often described by many other documents on its owner's host, and they comprise the second category. Generally, these RDF statements are also reliable. Some of them supplement its definitions (maybe inaccurate), while some others establish links from other SW objects to this one. So they often provide representative examples about how to use this SW object correctly. At last, all the others RDF documents belong to the third category. The RDF statements about a SW object from such documents are not quite reliable.

When constructing the summary of a SW object, we extract more RDF statements from the RDF documents in the first and second categories, and still extract a few ones from the third category. Because it is lacking of efficient ranking algorithms, the selection in each category is performed randomly. Nevertheless, it is observed that for most SW objects, all the RDF statements about them that can be extracted from the first and second categories are limited, so the summaries can cover sufficient critical RDF statements in most cases.

Finally, extracted RDF statements are clustered and then presented. Statements are put into the same cluster iff their predicates come from the same vocabulary. An exam-

ple summary of `http://www.w3.org/People/Berners-Lee/card#i` is depicted in Fig. 10. Each tab corresponds to a cluster, in which the statements often characterize a specific aspect of the SW object, e.g., about the contact information (FOAF) or about the project information (DOAP). Each RDF statement is also associated with its source.

7. RELATED WORK

TAP [10] is one of the earliest keyword-based Semantic Web search systems. It maps keywords to the labels of SW objects and selects a SW object based on the popularity, user profile, and search context. TAP extracts a subgraph from data and serializes it in the results. Swoogle [7], as one of the most popular Semantic Web search engines, provides services for searching terms (classes and properties) and ontologies. Terms and ontologies are ranked based on Swoogle's rational surfer model. Swoogle also provides their statistical metadata. SWSE [11] provides keyword-based search for SW objects. Similar to Falcons, SWSE enables users to filter the search results by specifying the classes collected from the results. To show a SW object, SWSE presents a list of RDF statements as well as their sources, and enables users to navigate related SW objects. Semantic Web Search [14] focuses on searching for specific types of SW objects, such as FOAF Person and RSS Item. The search results of this system are organized by documents so it cannot provide an integrated view of SW objects. Similarly, Watson [6] also organizes results by documents. Watson enables users to specify the scope that keywords should be mapped to, such as local names, labels, or any literals.

For keyword queries, most of these systems match query terms with the local names and associated literals of SW objects. Falcons expands the scope to also include the description of neighboring SW objects, so that it is more adaptable to keyword queries. Both SWSE and Falcons dynamically recommend classes for refining the search results. The distinguished features of Falcons Object Search is that it performs reasoning to obtain implicit types of SW objects, and uses hierarchy instead of list to manage the complexity of presenting tremendous classes.

8. CONCLUSION

To promote the development of a Web of data, developers are suggested to reuse existing URIs to identify things. To serve it, the Falcons system provides a keyword-based object search service for the purpose of reuse as well as the information needs. In Falcons Object Search, queries can be refined by navigating class hierarchies, integrated from distributed vocabularies on the Semantic Web.

The technical contributions of this paper are: a method to construct comprehensive textual description of SW objects; an approach to indexing SW objects from their classes and ancestor classes, including class subsumption reasoning on multiple vocabularies; a class recommendation technique to enable navigating class hierarchies; a preliminary summarization method for SW objects.

In future work, we aim to find or devise reliable ways to match the classes from different vocabularies. We are also working on an improved SW object summarization.

9. ACKNOWLEDGMENTS

This work is supported in part by the NSFC under Grant 60773106, and in part by the 973 Program of China under Grant 2003CB317004.

10. REFERENCES

[1] S. Abels, L. Haak and A. Hahn. Identification of common methods used for ontology integration tasks. In *Proc. IHIS*, pages 75–78, 2005.

[2] Z. Aleksovski, W. ten Kate and F. van Harmelen. Exploiting the structure of background knowledge used in ontology matching. In *Proc. OM*, pages 13–24, 2006.

[3] T. Berners-Lee, R. Fielding and L. Masinter. Uniform resource identifier (URI): generic syntax. *RFC 3986*, 2005.

[4] S. Chakrabarti, B. Dom, R. Agrawal and P. Raghavan. Using taxonomy, discriminants, and signatures for navigating in text databases. In *Proc. VLDB*, pages 446–455, 1997.

[5] S. Chakrabarti, B. Dom and P. Indyk. Enhanced hypertext categorization using hyperlinks. In *Proc. SIGMOD*, pages 307–318, 1998.

[6] M. d'Aquin, M. Sabou, M. Dzbor, C. Baldassarre, L. Gridinoc, S. Angeletou and E. Motta. WATSON: a gateway for the semantic Web. In *Proc. ESWC Posters*, 2007.

[7] L. Ding, R. Pan, T. Finin, A. Joshi, Y. Peng and P. Kolari. Finding and ranking knowledge on the semantic Web. In *Proc. ISWC*, pages 156–170, 2005.

[8] J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, Berlin Heidelberg, 2007.

[9] B.C. Grau, I. Horrocks, Y. Kazakov and U. Sattler. A logical framework for modularity of ontologies. In *Proc. IJCAI*, pages 298–303, 2007.

[10] R. Guha, R. McCool and E. Miller. Semantic search. In *Proc. WWW*, pages 700–709, 2003.

[11] A. Harth, A. Hogan, R. Delbru, J. Umbrich, S. O'Riain and S. Decker. SWSE: answers before links! In *Semantic Web Challenge*, 2007.

[12] D. Berrueta and J. Phipps. Best practice recipes for publishing RDF vocabularies. *W3C Working Draft*, 2008.

[13] S. Rajan, K. Punera and J. Ghosh. A maximum likelihood framework for integrating taxonomies. In *Proc. AAAI*, pages 856–861, 2005.

[14] Semantic Web Search. `http://www.semanticwebsearch.com/`.

[15] G. Stumme and A. Maedche. FCA-MERGE: bottom-up merging of ontologies. In *Proc. IJCAI*, pages 225–230, 2001.

[16] G. Tummarello, C. Morbidoni, R. Bachmann-Gmür and O. Erling. RDFSync: efficient remote synchronization of RDF models. In *Proc. ISWC*, pages 537–551, 2007.

[17] X. Zhang, G. Cheng and Y. Qu. Ontology summarization based on RDF sentence graph. In *Proc. WWW*, pages 707–716, 2007.

[18] D. Zhang and W.S. Lee. Learning to integrate Web taxonomies. *J. Web Sem.*, 2(2):131–151, December 2004.