10 Semantic Web Search Engine Revisited: A Prototype System

We discussed Semantic Web search engines in Chapter 2. More specifically, we dissected a search engine for the traditional Web and went on to show one possible way of making changes to this keyword-based search engine so that it could take advantage of the added semantics in the Web pages. This modified search engine could work as a Semantic Web search engine; however, it is far from optimal, as we will show in this chapter.

In fact, the Semantic Web search engine discussed in Chapter 2 represents just a minor modification of the current traditional search engine, and the reason we discussed it was not to propose a real Semantic Web search engine; we just used it as an example to show what the Semantic Web is and how it can help us, for example, to build a better search engine. We believe a direct comparison of a traditional search engine with a hypothetical Semantic Web search engine will give the reader an intuitive feel for the Semantic Web, a concept that can be very confusing for newcomers.

In this chapter, we will present a Semantic Web search engine in a much more systematic way, although we are not going to write code; but the design is ready, which is why we call it a prototype system. In fact, there is still no final "version" of any commercial Semantic Web search engine. It is our hope that after reading this chapter, you will be inspired to design a real Semantic Web search engine and make it a success; this could be the killer application of the Semantic Web vision that everyone is looking for.

10.1 WHY SEARCH ENGINES AGAIN

We have discussed the issue of semantic markup in Chapter 9. Semantic markup serves as the link between the two worlds: the world of the Web in its current state and the world of machine-readable semantics. As discussed previously, this is the actual implementation of "adding semantics to the current Web."

However, it seems none of the applications we have discussed so far really takes advantage of this link: Swoogle is a search engine for Semantic Web documents. The world of the Semantic Web, as far as Swoogle is concerned, is the world of Semantic Web documents, which includes ontology documents and RDF documents (SWDs). However, these documents only comprise one end of the link; it is fair to say that Swoogle completely ignores the other end of the link, namely, the current Web.

What about FOAF? The link in the FOAF project connects two subsets of both worlds: only personal Web pages from the world of the current Web are linked to the semantics about social life in the semantic world. Even this link does not play a vital role in FOAF: the main reasoning power of FOAF comes from aggregation of the markup files. Again, this concentrates on the semantic end of the link; the current Web at the other end of the link is not all that important.

What about search engines? How exactly will the Semantic Web vision help search engines? How should search engines take advantage of the added link? These questions remain unanswered. It is true, however, that the need for moving from the traditional Web to the Semantic Web arose mainly from the inefficiency of keyword-based search engines.

Therefore, let us return to the root of the challenge. In this chapter, we will look at how to improve current search engines by taking advantage of the available link between the two worlds. Again, what we will design in this chapter is just one possibility for such an improvement, and it mainly serves as an inspiration for possible better designs.

But first, before we start to come up with a new design, let us study the exact reasons behind the failure of keyword-based search engines.

10.2 WHY TRADITIONAL SEARCH ENGINES FAIL

From the discussion in the previous chapters, the answer to this question seems to be very clear already: each Web page is created mainly for human eyes, and it is constructed in such a way that the machine-understandable information on each page is just enough to tell the machine how to display the page, without understanding it. Therefore, a given search engine can do nothing about understanding the page and it is forced to take the keyword-based approach. When a user types keywords into the search engine, it will simply return all the pages containing the given keywords.

Let us go one step further by concentrating on the more fundamental reasons underlying the preceding obvious ones. As agreed upon by the research community, the following are these more basic reasons:

1. The exact same term can have different meanings.
2. Different terms can mean exactly the same thing.

These are the main reasons why we all have had very frustrating experiences with search engines: if the same term always has the same meaning and different terms always have different meanings, then even though the information on each page is only good for display, the keyword-based matching algorithm would still work very well.

Clearly, any possible design of a Semantic Web search engine has to take these two major difficulties into consideration; each has to be well addressed by the design to obtain a significant improvement in the performance of the engine.

10.3 THE DESIGN OF THE SEMANTIC WEB SEARCH ENGINE PROTOTYPE

To come up with a design of a search engine, we need to answer one by one the following questions:

- The discovery strategy: How are the documents discovered?
- The indexation strategy: How is the indexation done?
- The query-processing strategy: What does the user interface look like and how is the query processed?
- The result-screening strategy: How are the resulting pages sorted?

We will be addressing all these issues in the following sections, but let us first discuss the considerations pertaining to specific domains. As there will be ontologies involved in the search engine and ontologies are normally constructed for specific domains, what would be the underlying ontologies and domains for our search engine? For the following discussions, we will have one specific domain in mind (the camera ontology we developed, of course), but as you will see, the results and the methodology can be easily extended to include different or multiple domains represented by different ontology documents.

10.3.1 Query Processing: The User Interface

Let us discuss the user interface of the search engine first. The requirements/specifications of the search interface have a deep impact on the rest of the components of a search engine.

We have established the following basic rules regarding the search interface:

- The users must be able to simply type a few words into the search box to begin a search.
- The search result should be in the form of a set of links pointing to HTML pages.

Based on recently published research [44], empirical results from user testing of Web search systems show that most users simply type a few words (2–3) into the search box, and very few users actually use the advanced search options provided by the commercial search engines (such as Boolean searches, fielded searches, etc.). In our design, one important goal is to make the Semantic Web search engine feel just like the traditional one: the user does not need to know there is a semantic flavor added to the search engine. Any search queries based on the knowledge of the Semantic Web should not be necessary. For instance, the user should not have to construct the following search query:

```
[foaf:knows].[foaf:name]~"Liyang Yu"
```

This is very important to the users. Any query that has to be constructed using Semantic Web knowledge will simply scare or confuse the users away.

The second important consideration is the search results. Most users are used to the traditional search engines and they prefer the familiar HTML pages. It is the fact that so many irrelevant pages are included in the search results that frustrates them, not the pages themselves. After all, the goal of the search is to find relevant pages that they can read and explore. Swoogle, for example, mainly returns the ontology documents or RDF statement documents; these are good for Semantic Web researchers and developers, but these documents are mainly constructed for the machine to read, not for human eyes. In other words, these results mean almost nothing to casual users.

Having set up the design goal of the search engine user interface, all the rest will be guided by these criteria, as will be discussed next.

10.3.2 The Discovery Strategy: More Focused Crawling

Clearly, to improve the performance of the search engine, we have to take advantage of the fact that some Web pages are semantically marked up. Let us call these Web pages "semantically enhanced pages" (SEPs). More specifically, in our Semantic Web search engine, it is highly desirable that the crawler be able to discover and index as many SEPs as possible.

The most obvious design of the crawler is the normal one: the crawler is fed with some seed URLs, and it begins its journey from these seeds. More specifically, for every Web page it encounters, it will download the page and check the header part of the HTML code to see if this given Web page has been semantically marked up, i.e., whether the underlying page is an SEP. If so, it will download the markup document and also the ontology (or ontologies) involved and start to index the page carefully by taking advantage of the added semantics to ensure a performance improvement (we will talk more about this indexation procedure in great detail in the next section). Recall that this crawler pattern is exactly the one we discussed in Chapter 2.

However, the pattern proposed in Chapter 2 was merely for the purpose of showing how the Semantic Web could work for us; it will not work well in real life. The reason is simple: given the current status of the Web, discovering SEPs is just like searching for needles in a haystack; we will not get a sufficient number of hits. In fact, as we have mentioned in Chapter 2, based on its 2005 report, Google has indexed about 4–5 billion pages, but even these only account for about 1% of the pages contained in the Web. Given this vast number of pages, finding SEPs by using the normal crawling procedure is simply not practical.

Based on this observation, let us propose another way of crawling the current Web: let us call it a more focused crawler. Now, in order to illustrate how this crawler works, we need to limit the scope of the search engine to some specific domain. Let us assume that we are building a Semantic Web search engine just for the photography community and let us also assume that the only ontology that is widely accepted and used is the camera ontology that we developed in the previous chapters. As mentioned previously, the general workflow of the crawler can be easily extended to a much broader domain and easily modified to include more ontology documents.

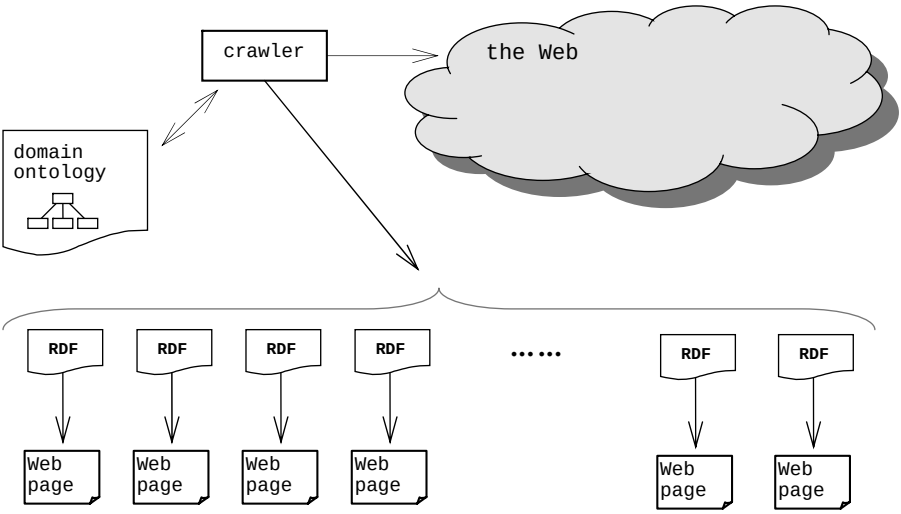

FIGURE 10.1 A more focused crawler.

Given this assumption, the following is the workflow of this more focused crawler (this process is described in Figure 10.1):

Step 1: Take the ontology document from the underlying domain and use Swoogle's APIs (a set of Web services) to find all the RDF instance documents created by using this given ontology.

Step 2: For a given RDF document discovered in step 1, decide whether it is a markup document for some Web page. If so, download that Web page and index it; otherwise, discard it.

Step 3: Repeat step 2 until every RDF document discovered in step 1 has been processed.

Step 1 will look familiar to you from Chapter 7. We have walked through an example of the same procedure; only, we did it manually. Swoogle also provides a set of Web services that we can use to implement the same function programmatically. In our case, we just need to call the right Web services to get all the RDF documents created by using our camera ontology.

Step 2 requires a little more work. The problem is, for a given RDF document, how do we know that it is a markup file for some Web page? If so, where can we find that Web page?

The easiest solution is to ensure that when the page is being marked up each markup document includes the lines shown in List 10.1.

Given these added lines (lines 2 to 5), the crawler will load the RDF document into memory, parse it, and look for the markup label and the URL label that points back to the Web page to which this markup document belongs. If no such label exists in the `owl:ontology` tag, the crawler will know this document is not a markup

LIST 10.1
The Markup Document Has to Refer Back to the Page Being Marked Up

```
1:  <?xml version="1.0"?>

2:  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
... the rest of the namespaces ... "/>

2:  <owl:ontology>
3:    <rdfs:label>markup</rdfs:label>
4:    <rdfs:label>{the URL of the Web page being marked up}</rdfs
          :label>
5:  </owl:ontology>

... the rest of the markup document ...
```

document for any Web page. Note that in the case where the original markup file also needs to include the `owl:ontology` tag, the two extra labels (lines 2 to 5) can still be added in without effecting the rest of the properties included in the `owl:ontology` tag.

Step 3 does not require much explanation. The obvious benefit of using such a crawler is that the maximum number of SEPs will be found and indexed. Here, we take advantage of the Swoogle system to implement this specific discovery process.

After we have collected all these markup documents, the next question is: how should we index these documents and the original Web pages associated with them?

10.3.3 The Indexation Strategy: Vertical and Horizontal

The indexation procedure conducted by the crawler can be described as two sub-procedures: the *vertical* indexation and the *horizontal* indexation. Let us take a look at the vertical indexation procedure first.

10.3.3.1 Vertical Indexation

For a given RDF markup document, the vertical indexation procedure can be described as in Table 10.1.

Let us use one example to illustrate exactly how these steps are followed. Suppose the crawler is currently examining the markup document shown in List 10.2, which is the document we created in the previous chapter, with the link pointing to the original Web page (as shown in Figure 10.1) being added in lines 6 to 9 (List 10.2).

By following step 1 in Table 10.1, the crawler has found the following two instances:

```
http://www.yuchen.net/pageMarkup.rdf#Nikon-D70
http://www.yuchen.net/people#LiyangYu
```

TABLE 10.1
Vertical Indexation Procedure

Step 1	Parse the rdf markup document to create a collection of all the instances described in this rdf document
Step 2	Take one instance from the collection created in step 1; do the following: 2.1 Find the class of this instance 2.2 Use the ontology to find all the superclasses of this class 2.3 Use the ontology to find all the equivalent classes of this class 2.4 Use the ontology to find all the equivalent classes of the superclasses discovered in step 2.2 2.5 Index all these classes
Step 3	Repeat step 2 until every instance in the collection has been processed

LIST 10.2
An Example Markup Document

```
1:  <?xml version="1.0"?>

2:  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:           xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:           xmlns:dc="http://www.purl.org/metadata/dublin-core#"
5:           xmlns:camera="http://www.yuchen.net/photography/Camera
                   .owl#">

6:  <owl:ontology>
7:     <rdfs:label>markup</rdfs:label>
8:     <rdfs:label>
          http://www.yuchen.net/Photography/NikonD70Review.html
       </rdfs:label>
9:  </owl:ontology>

10: <rdf:Description
        rdf:about="http://www.yuchen.net/Photography/NikonD70Review
            .html">
11:    <dc:title>D70 Review By Liyang Yu</dc:title>
12:    <dc:creator>Liyang Yu</dc:creator>
13: </rdf:Description>

14: <rdf:Description
         rdf:about="http://www.yuchen.net/pageMarkup.rdf#Nikon-D70">
15:    <rdf:type
        rdf:resource="http://www.yuchen.net/photography/Camera
            .owl#SLR"/>
16:    <rdfs:label>Nikon-D70</rdfs:label>
17:    <rdfs:label>D-70</rdfs:label>
18:    <camera:pixel rdf:datatype="http://www.someStandard.org
        #MegaPixel">
        6
```

```
19:     </camera:pixel>
20:     <camera:has_spec>
21:        <rdf:Description>
22:           <camera:model
                rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
                Nikon-D70
23:           </camera:model>
24:        </rdf:Description>
25:     </camera:has_spec>
26: </rdf:Description>

27: <camera:Photographer
              rdf:about="http://www.yuchen.net/people#LiyangYu">
28:    <rdfs:label>Liyang Yu</rdfs:label>
29: </camera:Photographer>

30: </rdf:RDF>
```

The crawler then moves on to step 2. It takes the first instance, and maps it back to its class type, i.e., the `camera:SLR`. It then traverses upstream in the camera ontology (the final version is shown in List 8.5), trying to find all the superclasses of `camera:SLR`. These superclasses are identified as `camera:Digital` and `camera:Camera`.

For any of these classes, the crawler then tries to find all their corresponding equivalent classes. In our case, `camera:Camera` is equivalent to `DigitalCamera`, and `camera:SLR` is equivalent to `SingleLensReflex`.

Now the crawler has collected the following classes: `camera:SLR`, `camera:Camera`, `camera:Digital`, `DigitalCamera`, and `SingleLensReflex`. It then indexes these classes as follows:

for each of one of these classes:
  the "local name" is used as indexing keyword;
  the value of `rdfs:label` is also used as the indexing keyword.

The local name of the class is defined as the substring after the "#" character in the URI of this class. For example, `camera:SLR`'s URI is given by

```
http://www.yuchen.net/photography/Camera.owl#SLR
```

Then the local name of `camera:SLR` is `SLR`, and this name is used as the indexing keyword.

As suggested previously, another indexing keyword is the value of the `rdfs:label` property. This is based on the fact that many ontology authors, when defining classes using OWL, choose to also include an `rdfs:label` property to describe another name of the underlying class. For the Semantic Web search engine, the significance of this value represents the fact that "several different terms/names can mean exactly the same thing;" this value and the local name of the class both refer to exactly the same concept, which is why the value of the `rdfs:label` property

is also used as an indexing keyword. In our example, none of the classes have the `rdfs:label` property defined, but it is not hard to see their importance, as has been described previously.

As this point, one index entry has been created for the original Web page: its URL is linked with the keyword `SLR`. Note that before this happens, the keyword `SLR` may already have a number of links to other pages, and the URL of our review page can also be linked by several other keywords. This relationship is shown in Figure 10.2.

The crawler then moves on to class `camera:Camera` and `camera:Digital`. Again, both of these classes' local names are used as keywords, but neither has `rdfs:label` property values to use as synonymous keywords. After these two classes are processed by the crawler, the indexation is updated as shown in Figure 10.3.

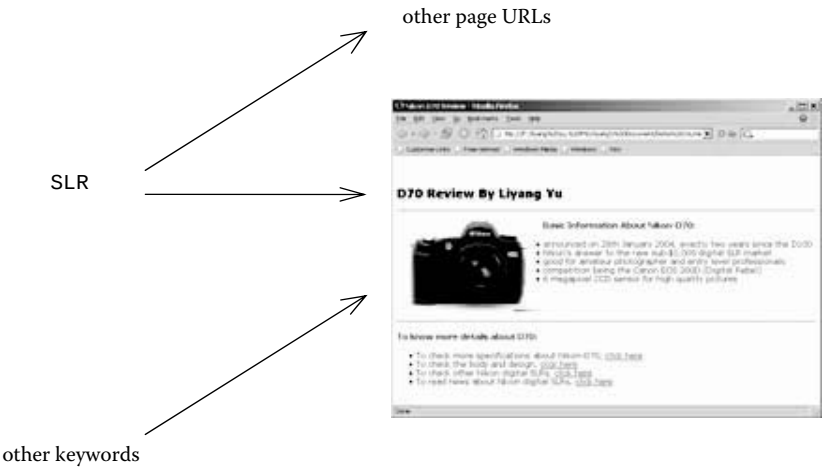

FIGURE 10.2 Index entry for our review page.

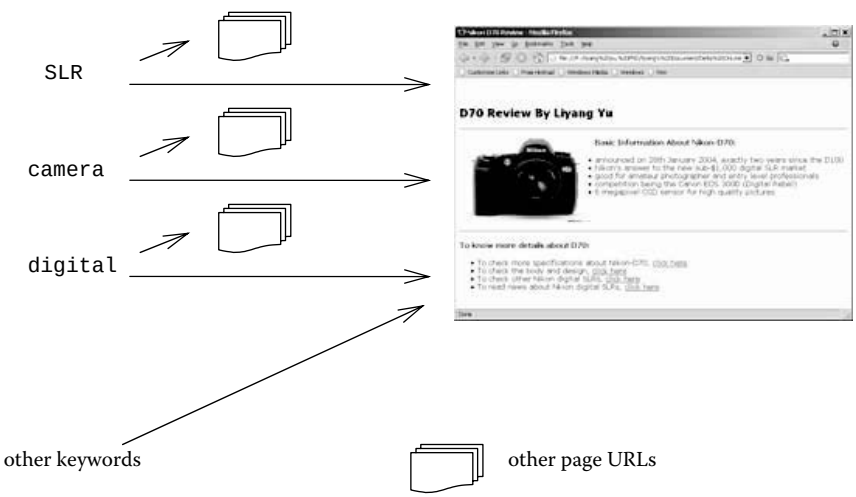

FIGURE 10.3 Updated indexation system (1).

You can imagine what the indexation system would look like after the crawler has processed `DigitalCamera` and `SingleLensReflex` class. The key point is that by traversing the ontology document, all the related terms, i.e., the local names of superclasses, are used as synonymous keywords by the crawler to update the index. The goal is to express the fact that different terms can have exactly the same meaning.

The reason why this indexation procedure is called vertical indexation should be clear by now: after we identify the class of the current instance, the ontology is traversed vertically (upstream) to find all the superclasses and the equivalent classes to add more synonymous keywords.

At this point, the crawler finishes handling the first instance in the markup document and reaches step 3. Step 3 sends the crawler back to step 2, with the next instance, i.e., `http://www.yuchen.net/people#LiyangYu`. The crawler first concludes that this instance has a class type of `camera:Photographer`, and as it did for the first instance, it further identifies the related class set that contains the following classes: `camera:Photographer` and `foaf:Person`.

Again, class `camera:Photographer` does not have an `rdfs:label` property, but `foaf:Person` does, and this property also has a value `Person`, which is identical to the local name of the class; therefore, in this case the `rdfs:label` property value does not add any new synonymous keyword.

At this point, the indexation system is shown in Figure 10.4. Note that for simplicity, the keywords from `DigitalCamera` and `SingleLensReflex` are not included.

As far as this example goes, the crawler has finished the process of vertical indexation, during which several related terms have been added to the indexation as synonymous keywords.

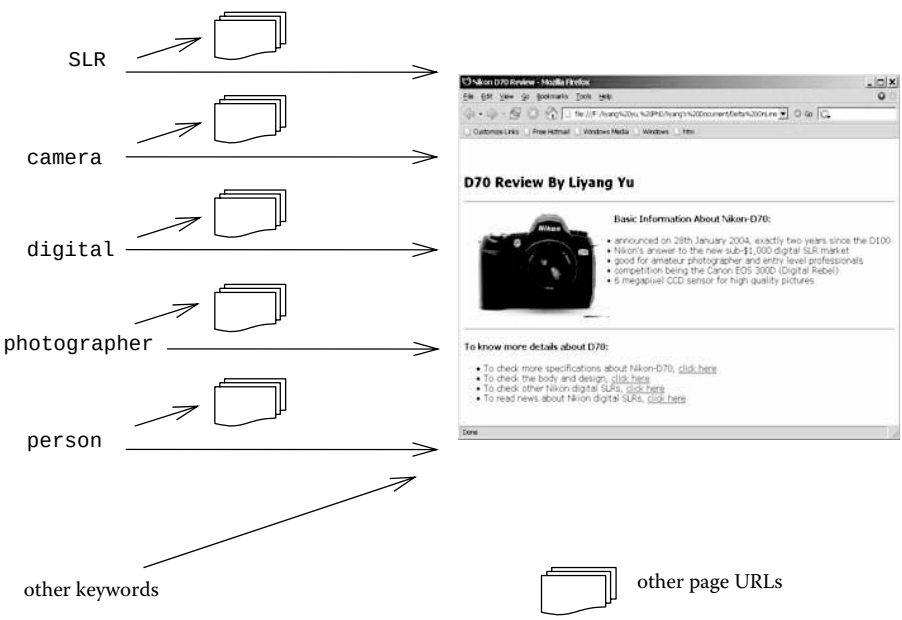

FIGURE 10.4 Updated indexation system (2).

10.3.3.2 Horizontal Indexation

At this moment, no indexation has yet been done on the markup document itself. In this section, we will see how the indexation is done on the markup document shown in List 10.2. This indexation process is called *horizontal indexation.*

This straightforward process is summarized in Table 10.2.

Note that after List 10.2 is parsed, it contains the following triples shown in Table 10.3. (It is easier to show the indexation process using this triple format. You can manually generate these triples by using the validator discussed in Chapter 3.)

Horizontal indexation is conducted by scanning this triple set. The crawler fetches the first instance, `~NikonD70Review.html`, and identifies that this is the

TABLE 10.2
Horizontal Indexation Procedure

Step 1	Parse the RDF markup document to create a collection of all the instances described in this RDF document
Step 2	Take one instance from the collection created in step 1; do the following: 2.1 Index the local name of this instance 2.2 Find all the instance-level `rdfs:label` property values and index them 2.3 For every user-defined property, index this property's local name and `rdfs:label` value found in the ontology document where this property is defined 2.3.1. If this property uses a typed or untyped literal as its value and this value is a string (nonnumerical), index this value 2.3.2. If this property uses another instance as its value, add this instance to the instance collection created in step 1
Step 3	Repeat step 2 until every instance in the collection has been processed

TABLE 10.3
List 10.2 Expressed in Triple Format

Subject	Predicate	Object
~NikonD70Review.html	dc:title	D70 Review By Liyang Yu
~NikonD70Review.html	dc:creator	Liyang Yu
markup:Nikon-D70	rdf:type	camera:SLR
markup:Nikon-D70	rdfs:label	Nikon-D70
markup:Nikon-D70	rdfs:label	D-70
markup:Nikon-D70	camera:pixel	6
markup:Nikon-D70	camera:has_spec	genid:ARP4420
people:LiyangYu	rdf:type	camera:Photographer
people:LiyangYu	rdfs:label	liyang yu
genid:ARP4420	camera:model	Nikon-D70

```
~: http://www.yuchen.net/Photography/
markup: http://www.yuchen.net/pageMarkup.rdf#
people: http://www.yuchen.net/people#
dc: http://www.purl.org/metadata/dublin-core#
camera: http://www.yuchen.net/photography/Camera.owl#
```

metadata created by using Dublin Core; therefore, every word in `dc:title` and `dc:creator` will be used as an index word. At this moment, "`D70`," "`Review`," "`Liyang`," and "`Yu`" are added to the index table.

The second instance the crawler comes across is `markup:Nikon-D70`. The crawler first adds the local name, "`Nikon-D70`," to the index table. There are two instance-level `rdfs:label` properties, one having the value "`Nikon-D70`" and the other having the value "`D-70`," both of which are used as keywords in the index table. After this step, two new keywords have been indexed in the table: "`Nikon-D70`" and "`D-70`."

The crawler then proceeds to the first user-defined property, `camera:pixel`. The local name of this property, `pixel`, is added to the index table. The crawler accesses the ontology document containing the definition of this property to locate the `rdfs:label` value in the definition. As no `rdfs:label` property has been declared in the definition of `camera:pixel`, the crawler returns with a `null` value. Note that the value of `camera:pixel` property is `6`, a nonstring typed value; therefore, the crawler ignores it. After this step, `pixel` is used as the next keyword for the indexation.

The second user-defined property is `camera:has_spec`. The local name `has_spec` is added to the index table, and because there is no `rdfs:label` property defined for this property, no other synonymous keyword has been added. However, something interesting does happen here: the value of this property is another instance called `genid:ARP4420`, and the crawler adds this instance to the collection of all instances and moves on to the next step.

Up to this point, the crawler has finished processing `markup:Nikon-D70`. The next instance in the instance collection is `people:LiyangYu`. Its local name, `LiyangYu`, is added into the index table, and its `rdfs:label` value in this case is "`liyang yu`," which has already been indexed.

The last instance, `genid:ARP4420`, is fetched from the instance collection. The crawler first tries to index its local name. However, this instance does not have a local name, because it is an anonymous instance whose type is `camera:Specifications`. The crawler "understands" this (because of the help from a parsing and reasoning tool, for instance, `Jena`), so it adds the local name of the class to the index table. In this case, `specifications` are added. One user-defined property is associated with this instance, and its local name, `model`, is added to the index table. Its value, "`Nikon-D70`," was previously added and so this value is ignored.

At this point, the instance collection becomes empty and the crawler has finished the horizontal indexation process on the markup document. The current indexation system related to this review Web page is shown in Figure 10.5.

One last question we need to address is the issue about whether we need to index the original Web page. The answer is no. And the reasons are as follows:

- The reason why a Semantic Web search engine should deliver a better performance than a traditional search engine is because it uses the added semantics to index and discover highly relevant Web pages.
- The semantics of the relevant Web pages is assumed to be captured in their corresponding markup documents.
- The "unmarked-up" information left in the original Web page is assumed to be unrelated to the given semantics.

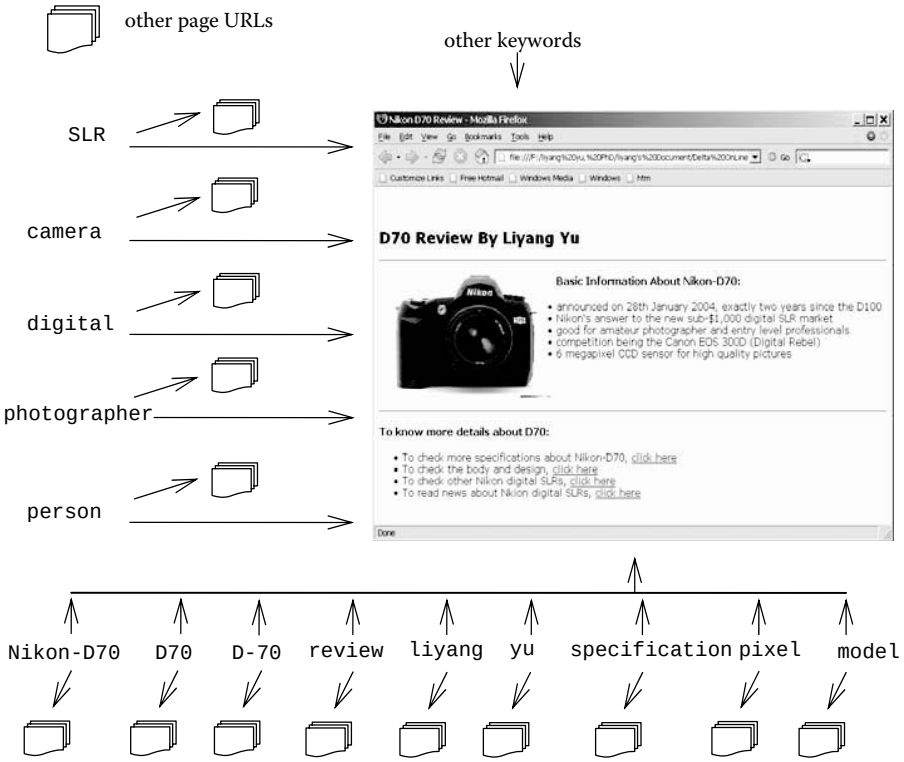

FIGURE 10.5 Final indexation system.

Therefore, there is no further need to index the original Web page. To a great extent, this decision makes the markup document crucial:

- When you create the markup document, you need to capture carefully all the important information contained in your Web page.
- If the ontology or ontologies you have selected cannot offer you enough semantic expressiveness to markup the vital information in your Web page, you need to choose another ontology or other ontologies, or even come up with your own ontology.

Again, this decision and all the previous decisions we have made, including even this prototype itself, are just here to illustrate how exactly the added semantics help us come up with better designs. There certainly is room for further improvement and development. If you are inspired to pursue this route further, then we have achieved our goal.

In the rest of this chapter, we will take a look at how to use this prototype and then we will discuss some more issues, one of them being the question of why this design is likely to provide better performance.

10.4 USING THE PROTOTYPE SYSTEM

As we have mentioned before, one design goal is to keep the user interface as traditional as possible: the user should not need to construct complex queries and the returned results are links to the Web pages. However, there is something extra the users have to do: they have to specify the underlying ontology before they can start the search.

The goal of specifying the ontology is to clearly indicate the context of the current search. Imagine a scenario in which there are two different camera ontologies constructed by two different groups of domain experts. One group of experts is mainly interested in the sale of cameras, so that information such as price, inventory level, warehouse location, order fulfillment, etc., constitute the main concepts in their camera ontology. The other group of experts is interested more in the performance and specifications of different cameras. Most of these experts probably come from a photography background and the ontology they constructed is for the evaluation and recommendation of cameras to different levels of photographers. The camera ontology we have been developing throughout this book is one example of such an ontology.

When a page owner is marking up his page, the first thing he or she needs to decide is what ontology to use. If the Web site is mainly for camera sale, the page owner would decide to use the first camera ontology. In our case, recall that our Web page is mainly for evaluating a given camera; therefore, we have chosen the second ontology.

The conclusion is that the semantic context of each Web page should be well defined at the time this page is being marked up. The indexation process, as we have described in the previous section, solely depends on the markup document and the underlying ontology. The final result is that a certain number of keywords are linked to a Web page in a well-defined semantic context. When a search is about to begin, the search engine has to be informed about this well-defined semantic content.

Suppose I want to search for "SLR" to learn more about its performance, its targeted users, and other related information to decide if it is the best choice for me. Therefore, I am not interested in knowing the available vendors and prices (although I will be interested in this later on). I then clearly indicate this context to the search engine by specifying the ontology. With this vital information, the search engine is able to return the most relevant results, given the way the documents are discovered and how the indexation system is constructed.

How do we tell the search engine which ontology is being specified? A possible solution is to add a drop-down list box beside the query textbox; clicking the list box will show all the available ontologies, and the user can choose the one he or she is interested in. This will require a certain degree of familiarity with the ontologies.

To go one step further, imagine a categorization schema that partitions the entire Web into several different major domains, such as education, medicine, entertainment, finance, etc., and for each domain, assume that several major ontologies have been constructed by a group of domain experts.

Now, use our prototype search engine to build the index tables. It will take the first domain, find all the ontologies related to it, and discover on the Web all the

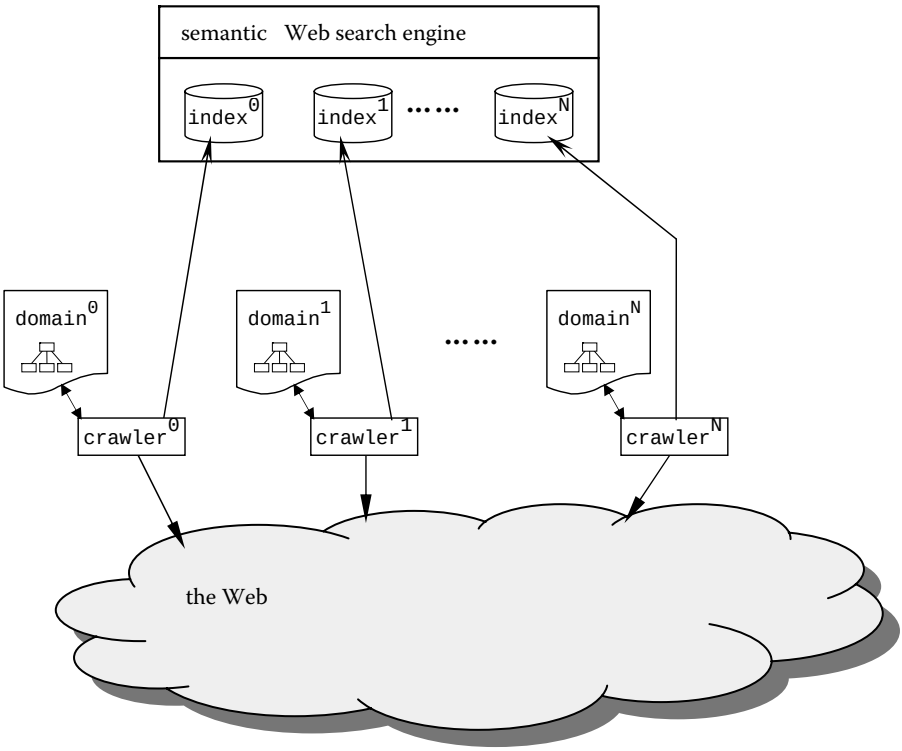

FIGURE 10.6 The indexation process given multiple domains.

pages that have been marked up using one of these ontologies. It will then continue to build the corresponding index table for this domain just as we have discussed in the previous sections. After all the documents have been indexed, the search engine will move on to the next domain to build the next index table, and so on and so forth, until all the domains are covered. This is in fact one possible way to extend our prototype system to make it a really "big" search engine (see Figure10.6). Note that there are multiple index tables inside the search engine database: one domain now has its own specialized index table.

Given this structure, the user is not required to specify the ontology against which the search will be conducted. All that is needed is to select a domain before a search is launched. Once a domain is specified, only the index table belonging to the specified domain will be used to come up with the resulting pages.

10.5 WHY THIS PROTOTYPE SEARCH ENGINE PROVIDES BETTER PERFORMANCE

Now that we understand the structure of the prototype search engine and other issues related to this engine, it is time for us to examine the key question: why does this search engine provide better performance than the traditional search engines?

To answer this question, recall the main reasons why a traditional search engine has trouble with search results. To recapitulate, here are the reasons:

1. The same terms can have different meanings.
2. Different terms can have the same meaning.

To deliver better performance, we need to ensure these two problems are addressed explicitly.

The first difficulty is overcome by the introduction of the concepts of domain and domain ontology into the design of our prototype, and the ontologies are explicitly accessed during the process of marking up Web pages and building the index tables. More specially:

1. An ontology is created to define the semantics of each concept (classes or properties) without any potential ambiguity.
2. Each keyword in the index table must be derived from one of the following:
   - The local name of an instance of some class
   - The local name of a property
   - The string value of a given property
   - Class names collected by vertical indexing
3. Because the semantics of all the classes and properties are clearly defined in the ontology, keywords based on these classes and properties also have the right contextual meanings.
4. A user who employs this search engine agrees with the semantics expressed in the ontology (he or she may have to specify which ontology to use before starting the search, as discussed in the previous section). In other words, the implied context from the user agrees with the context defined by the ontology.

The final result is that any given keyword in the index table only processes a single well-defined meaning, thereby eliminating the possibility that the same keyword can have different meanings.

An obvious drawback is that the search engine has to be specialized for some given domain. However, this is not a significant hurdle. Given what we have discussed in the previous section, we can come up with a predefined categorization of domains and have an index table for each domain (Figure 10.6). More specifically, the same word, such as "SLR," can appear in multiple index tables with quite different contextual meanings. As the user has to choose the domain before the search is launched, the meaning of "SLR" is well defined when the search is being conducted.

The second difficulty appears to be harder to address. For example, when searching for the Nikon D70 camera, one user may choose "D70" as the keyword, whereas another may use "D-70"; however, these different terms have the same meaning.

Generally speaking, these terms are quite often closely related and can be considered synonymous terms. Capturing this relationship appears to be part of the

solution to this problem. In the prototype search engine, the following is done to facilitate the handling of this problem:

1. Vertical indexation is implemented to collect all the conceptually related terms. As we have seen in the previous sections, the crawler will access the ontology and gather the local names of the superclasses. For instance, the local name "SLR" will trigger the crawler to also collect the local names such as "Digital" and "Camera." The rationale behind this is the fact that as these classes are conceptually related, their local names can naturally be treated as similar terms.
2. As the `rdfs:label` is often used by the ontology developers and markup document creators as a natural language complement of the local name, the value of the `rdfs:label` property can also be treated as a synonymous term to the local name of the class or property. As you have seen, the prototype search engine is designed to take advantage of this feature to capture the fact that different terms are used to reference the same concept.

These are clearly some heuristics developed to handle this situation. Again, our hope is that they can serve as some hints for designing better solutions.

Also, note that these heuristics have transferred some of the burden to the ontology developers and markup documents authors. Carefully using the `rdfs:label` property to include as many synonymous terms as possible is therefore an effective way to improve the hit rate of a given Web page. In fact, syntactically there is no limitation to how many `rdfs:label` properties you can use to describe a given instance or class. For example, List 10.3 has used several `rdfs:label` properties to describe a `Map` instance.

LIST 10.3
Using `rdfs:label` as Many Times as You Want to Add Terms for the Same Concept

```
<rdf:Description rdf:about="http://www.yuchen.net/example.rdf#someMap">
   <rdf:type rdf:resource="http://www.yuchen.net/example.owl#Map"/>
   <rdfs:label>map</rdfs:label>
   <rdfs:label>chart</rdfs:label>
   <rdfs:label>atlas</rdfs:label>
   <rdfs:label>drawing</rdfs:label>
   <rdfs:label>diagram</rdfs:label>
   ...
</rdf:Description>
```

These RDF statements enumerate some common words for mapping by using the `rdfs:label` property. When a crawler reaches this document, it will collect all these terms, hoping to be able to cover the keyword a given user could pick when conducting a search. It is not hard to imagine that it is quite possible a particular

search based on some keywords will return pages that do not contain any of the given keywords at all.

10.6 A SUGGESTION FOR POSSIBLE IMPLEMENTATION

We have presented a prototype for the Semantic Web search engine in this chapter. However, we did not mention the implementation details. In this last section, we briefly discuss some possible ways of testing this idea.

To construct a very simple and small test system, one can use Java and Jena as the development tools and treat your local hard drive as the Web world to build both a traditional search engine and the Semantic Web search engine that we have discussed in the chapter.

Preparation: Ensure you have some HTML documents on your local hard drive. Several of these should be pages describing digital cameras; some pages should contain information about camera prices, vendors (store locations, phone numbers, e-mails, etc.), and others should be about camera evaluations, performance comparison, etc.

Build a traditional search engine: Use Java to write a crawler to travel the directory tree structure, find all the HTML documents, and implement a full-text indexation on these documents. The index table should be created and maintained by using some database system. Then build a simple search interface to conduct a search. For instance, search for "SLR." You will probably see all the camera pages are returned, including both the sales pages and the evaluation pages.

Build our Semantic Web search engine: Now, use the camera ontology to markup all the pages containing the evaluation information, and save all the markup documents on your local hard drive. Rewrite your crawler to make it work like our prototype presented in this chapter. To do so, you will need to use Jena APIs to facilitate the horizontal and vertical indexation procedures and other necessary reasoning processes. After you are done, you will have an index table that looks quite different from the previous one. Now conduct a search using "SLR" as keyword (or other keywords), and you will see the difference: only the relevant pages are returned, and you do not have to sift through the pages again.

I strongly recommend that you do this exercise, because you will gain a better understanding of both traditional search engines and Semantic Web search engines. More importantly, you will begin to appreciate the power of the Semantic Web.

Part 4

From The Semantic Web to Semantic Web Services

Congratulations on going this far into our book. Having seen so much value added by the Semantic Web vision, you might have started wondering already about all that the Semantic Web can do for Web services.

Your intuition is right: adding semantics to Web services will change the way you use these services in your applications, and more specifically, the goal is to automatically discover the requested service, invoke it, composite different services to accomplish a given task, and automatically monitor the execution of a given service.

In this part, we will concentrate on automatic service discovery. To accomplish this goal, we will repeat the whole cycle again: the first step is to introduce the concept of Semantic Web services. We accomplish this by carefully reviewing the current Web service standards including WSDL, SOAP, and UDDI, and this discussion will enable us to clearly realize the need for Semantic Web services. The next step is to describe the technical details: We will study the upper ontology for service descriptions, and we will also study a markup language (OWL-S) in great detail so you can use it to describe the semantics of a given service. The third step is to show the how-to part: Given the OWL-S markup language and the current Web service standards, we will demonstrate exactly how we change a traditional Web service into a Semantic Web service and also map these descriptions to the current service standards. After this step of the cycle, the entire picture will become very clear.

To put all of these together, and to accomplish the goal of automatic service discovery, the last step of the cycle is to see a real example: We will develop a search engine for Semantic Web services using Java and Jena APIs. The benefit of

implementing this system is to review everything you have learned and also to acquire the main skills needed for developing Semantic Web applications. For you, this is probably the most exciting part!