2 Search Engines in Both Traditional and Semantic Web Environments

As we discussed in Chapter 1, before we get into the nuts and bolts of the Semantic Web, let us first understand how a search engine works. More specifically, in this chapter we are going to study the behavior of a search engine in both the traditional Web environment and Semantic Web environment. This detailed study and comparison will enable us to see the clear benefits offered by the Semantic Web, and it will also show some important components that are vital in the world of the Semantic Web. This will naturally provide us with a solid foundation for going further. Understanding the major components in the Semantic Web world and realizing how they fit together will make our life much easier. The technical details presented in the next several chapters will all seem to be a natural extension of the current Web, and this is exactly how the Semantic Web should be comprehended.

2.1 SEARCH ENGINE FOR THE TRADITIONAL WEB

Today, the most visible component of the Internet is the Web. It has hundreds of millions of pages available, presenting information on an amazing variety of topics — in fact, you can find almost anything you are interested in on the Web, and all this information originates from search engines.

In the current world, there are many different search engines you can chose from. Perhaps the most widely used ones are Google and Yahoo! There are also other search engines such as AltaVista, Lycos, etc. However, in some sense, all these search engines are created equal, and in this section, we are going to study how they are created.

More importantly, remember the frustration each one of us has experienced when using a search engine? In this section, we will begin to understand the root cause of this frustration as well. In fact, based on the discussion in this section, you can even start building your own search engine and play with it to see whether there is a better way to minimize the frustration.

2.1.1 Building the Index Table

Even before a search engine is made available on the Web, it starts preparing a huge index table for its potential users. This process is called the *indexation* process, and

it will be conducted repeatedly throughout the life of the search engine. The quality of the generated index table to a large extent decides the quality of a query result.

The indexation process is conducted by a special piece of software usually called a *spider*, or *crawler*. A crawler visits the Web to collect literally everything it can find by constructing the index table during its journey. To initially kick off the process, the main control component of a search engine will provide the crawler with a *seed* URL (in reality, this will be a set of seed URLs), and the crawler, after receiving this seed URL, will begin its journey by accessing this URL: it downloads the page pointed to by this URL and does the following:

Step 1: Build an index table for every single word on this page. Let us use URL_0 to denote the URL of this page. Once this step is done, the index table will look like this (see Figure 2.1).

It reads like this: word_1 shows up in this document, which has a URL_0 as its location on the Web, and the same with word_2, word_3, and so on. But what if some word shows up in this document more than once? The crawler certainly needs to remember this information. Therefore, an improved version of the initial index table will look like the one shown in Figure 2.2.

Now, we can read the index table like this: word_i is mapped to an object of "location structure," which has two pieces of information: the first piece of information says word_i shows up in a document located at URL_0, the second piece of information tells us word_i shows up in this document $c_i \geq 1$ times.

At this moment, it seems that the crawler has collected a fair amount of information about this current page. It is time to move on.

Step 2: From the current page, find the first link (which is again a URL pointing to another page) and "crawl" to this link, meaning to download the page pointed to by this link.

Step 3: After downloading this page, start reading each word on this page, and add them all to the index table.

Once the crawler reaches the new page by following the link on the current page, it will again start reading each and every word on this new page and add all the words to the index table. Now there are two possibilities: (1)

word_1	URL_0
word_2	URL_0
⋮	⋮
word_N	URL_0

FIGURE 2.1 The initial index table built by the crawler.

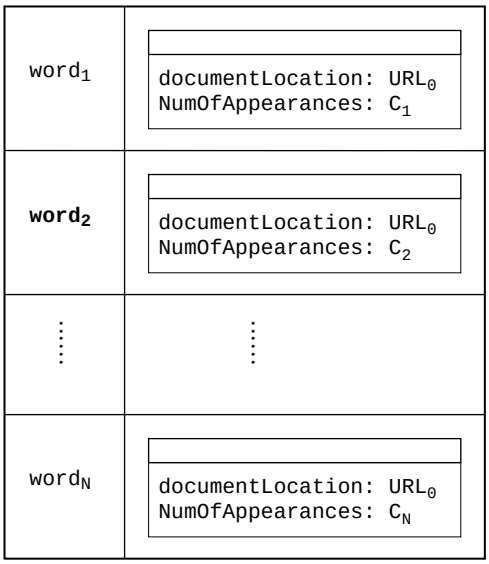

FIGURE 2.2 An improved version of the initial index table.

the current word from this new page has never been added to the index table, or (2) it already exists in the index table.

The simple case is the case where the word has never been added to the index table; we just need to add it and the newly added item looks just like any row in Figure 2.2. If the word (for example, $word_2$ shows up again in this new page) has already been added to the index table, then a little more work needs to be done: locate the word in the index table and grow its location structure, as described in Figure 2.3.

Using $word_2$ as an example, the index table is now read like this: $word_2$ shows up in both pages. In the first page, it shows up C_2 times, and in the second page (pointed to by URL_1), it shows up C_{1-2} times. Also, note that the length of the index table now changes to `N + M`, signifying that new words were found in the current document page.

Now the crawler finishes the second page as usual; to crawl all the pages, it continues its journey by going to step 4.

Step 4: Go to step 2, until no unvisited link exists (more about this later). After steps 2 and 3 have been repeated many times, the index table will look like the one shown in Figure 2.4, which is interpreted in the same way you understand Figure 2.3.

Steps 1 to 4 describe the basic flow of the crawler; it is one of the possible ways of visiting the whole Web. In reality, however, steps 2 and 3 will never be finished, because of limited resources (memory, time, etc.). As reported by a recent article in *Federal Computer Week* [31], Google, one of the most popular search engines, at best can index about 4 billion to 5 billion Web pages, representing only 1% of the World Wide Web (www).

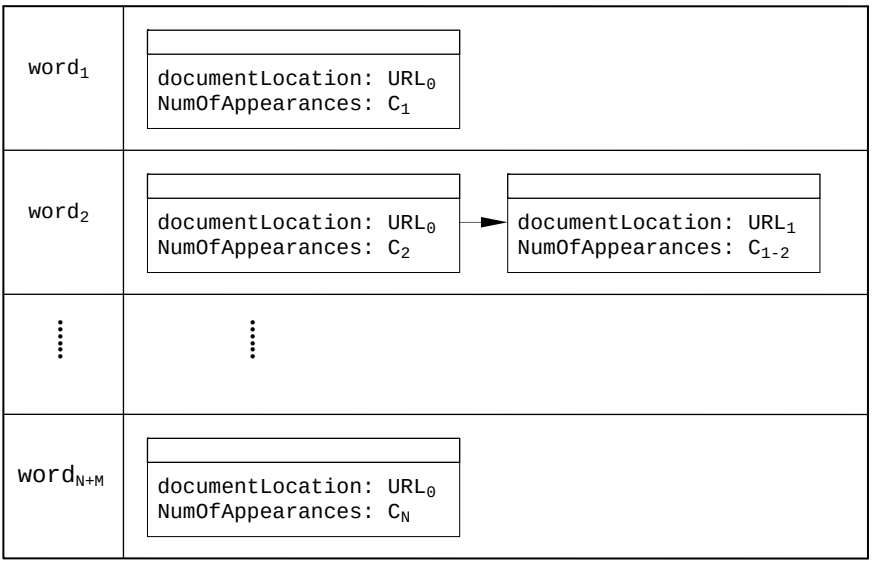

FIGURE 2.3 $\texttt{Word}_2$ in index table shows up on two pages.

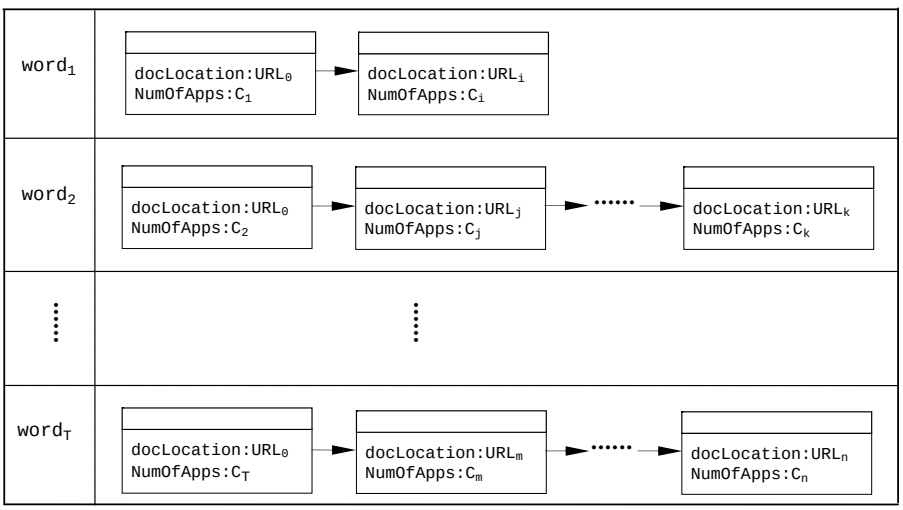

FIGURE 2.4 Final index table.

2.1.2 Conducting the Search

Now that the crawler has prepared the index table, a user can start a search. The simplest query is a single word, such as $\texttt{word}_2$. What if $\texttt{word}_2$ is never collected into the index table, meaning none of the documents the crawler ever visited has $\texttt{word}_2$ in it? The search engine simply returns a message such as "no results have been found for your query." In our case, $\texttt{word}_2$ is indeed in the index table; the search engine then iterates through the document records (as shown in Figure 2.5) and

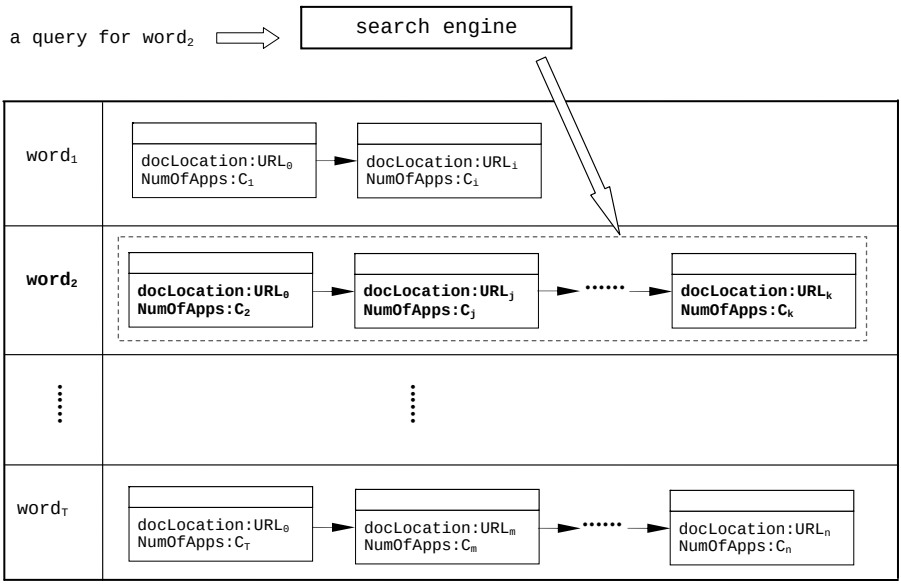

FIGURE 2.5 Search engine locates the document records for querying $word_2$

returns them in a particular order to the user, who can then click on each individual URL to get to the real Web page.

In what particular order are the document records searched? Given the structure of our index table, each document containing $word_2$ inside it also has a number associated with it that indicates how many times the word has turned up in this document. Therefore, the search engine with this index table can only sort the document according to this number: the document in which $word_2$ shows up the most number of times will appear at the top of the list returned to the user.

This is where the concept of *rank* kicks in: when returning the pages to the user, the search engine has to decide in which order they should be presented. You might also wonder what else is important for the quality of a search engine. Well, you are right: the search engine we just presented is in its simplest form; it will work if you code it, but there is a lot more to it.

2.1.3 Additional Details

In the simplest case, a search engine could just index the word and associate it with the URLs where it was found. However, this makes the engine not quite useful as there would be no way of telling whether the word was used in an important or an incidental way on the page. For instance, the word could be used just once in the document or it could be used many times. To be more specific, there has to be some mechanism so that a ranked list can be returned, with the most useful page at the top of the list.

As you can tell, in the search engine we just presented, we store the number of times the word appears on a page. Based on this number, our engine is able to assign

a weight to the page; the more often the word appears on the page, the more relevant this page is.

The next step is to remember where the word appears. It could appear in the title of the page, near the top of the page, in subheadings, in links, or in the metadata tags (you now realize why metadata is important). Clearly, these different locations of a given word signify the different levels of importance of the word. A much more complex ranking system could be built on this consideration and, in fact, each commercial search engine has a different way of assigning ranks to pages, the design and implementation details of which are normally not available to the public. Nevertheless, this is one of the reasons why a search using the same keyword on different search engines will normally produce different lists (the pages are shown in different orders).

As we are on the topic of page ranking and as we have already seen that metadata could be useful even in the current Web world, let us take a closer look at the role of metadata in these traditional search engines.

The key point to remember is that metadata can play a unique role in guiding how the crawler should build the index table. For instance, we have learned that Dublin Core (DC) elements, as one form of metadata, can be embedded into a given page. One of these DC elements, `Subject`, describes the topic of the page and, typically, it is expressed as keywords or phrases that describe the subject or content of the page. Assuming the words contained in this element are the important ones on this page, the crawler may decide to assign greater weight to these words, which would further change the rank of this page.

Another example is when the page owner decides to use formal classification schemes in the `Subject` element, indicating the general topic of the page (Entertainment, Business, Education, etc.). As it is also common for a given word to appear several times on the same page and with a different meaning each time, a smart crawler, on reaching this word, can use the classification information embedded in the metadata to decide what might be the most likely meaning of the word. On the other hand, it is also possible that the owner of the page added `<meta>` tags indicating the specific topic covered by the page but, in fact, the page has nothing to do with that topic. The result would be a wrong ranking number that might favor the page owner.

Metadata plays a vital role in search engine implementation: this will become more obvious when we examine how the search engine works in the Semantic Web environment. We will cover these interesting and exciting topics later. Let us continue our study of the traditional search engines; there is still a lot more to learn.

Recall that the crawler in our search engine begins its travels on the Web by following a seed URL. It turns out that this is the simplest task. In reality, you can prepare a list of seeds for the crawler to start (and usually these seeds are all popular sites), and you can even access a domain name server (DNS) to get a list of starting URLs. Also, because the crawler will never be able to conquer the whole world, you might also want to feed it a list of URLs that you do not want it to miss. Clearly, there are many ways to improve the performance of a search engine.

Another topic is how the Web world is visited. Recall the way our crawler works: it starts from the seed URL, finishes with the page content and identifies all the links on this page, and follows the very first link to the next page. It then repeats the

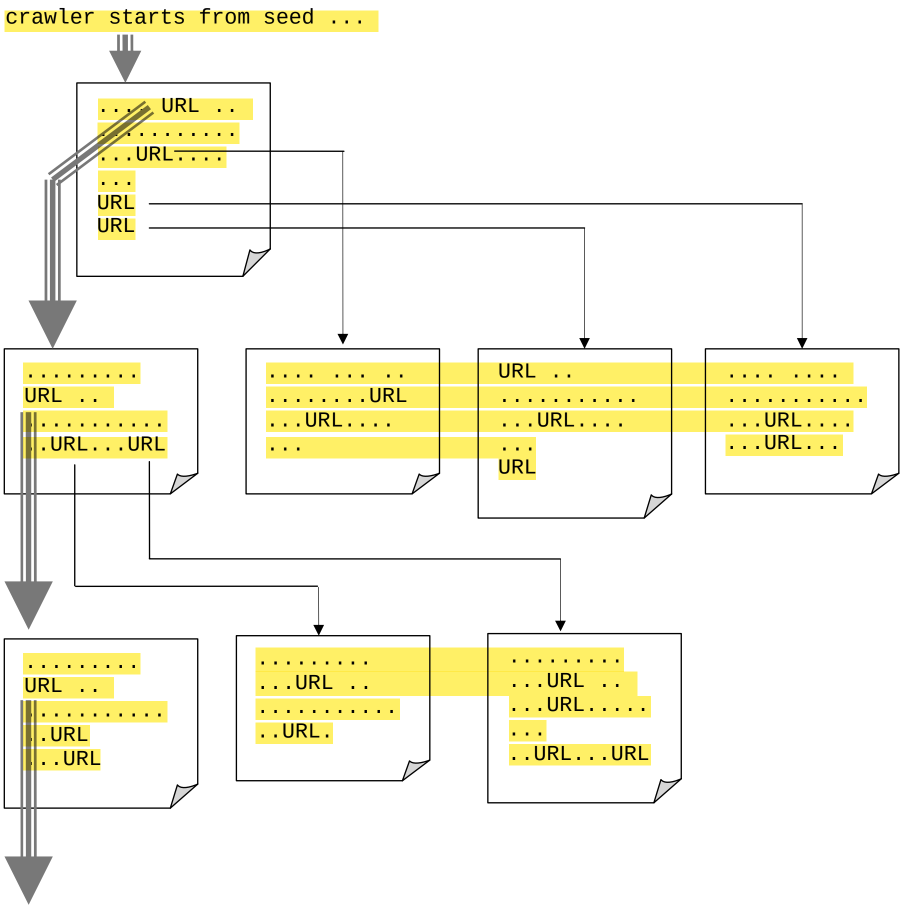

FIGURE 2.6 Depth-first search conducted by the crawler.

whole process on the second page before moving to the third page, and so on. This process is shown in Figure 2.6. The heavy arrow shows the movement of the crawler. This type of search is called *depth-first search*.

The counterpart of depth-first search is *breadth-first search*. I leave it to the readers to picture how a breadth-first search crawler works. Which one is better? I will leave this also for you to answer. Once again, you can see how much interesting work can be done in this area.

There are many other topics to discuss regarding search engines. Let us not go into further details, as the reason we study it is to gain a better understanding of how a search engine would work in a Semantic Web environment. Let us summarize the key points about search engines under the traditional Web:

- The Web is made up of billions of Web pages.
- Most Web page have three different kinds of codes: HTML/XHTML tags (required), CSS tags (optional), and JavaScript (optional).

- All these tags in a Web page just tell the machines how to display the page; they do not tell the machines what they mean.
- When a crawler reaches a given page, it has no way to know what this page is all about. In fact, it is not even able to tell whether the page is a personal page or a corporate Web site.
- Thus, the crawler can only collect keywords, turning all search engines into essentially keyword matchers.

2.2 SEARCH ENGINE FOR THE SEMANTIC WEB: A HYPOTHETICAL EXAMPLE

In this section we are going to show a hypothetical search engine example in the Semantic Web environment. The purpose is to let the reader understand precisely what Semantic Web is and what value can be added by extending the current Web to the Semantic Web.

Up to the time of this writing, the author has never encountered any real-world example of semantic search engines, although it is quite a popular research topic in the Semantic Web research community. Therefore, the example presented here can be viewed as one possible way of constructing a Semantic Web search engine. Again, the goal of examining such a hypothetical model is to gain a greater understanding of what the Semantic Web is.

2.2.1 A Hypothetical Usage of the Traditional Search Engine

To construct a search example (and the example is also going to be used throughout the book), let us start with my hobby, photography (which also happens to be a very expensive hobby). Similar to many other amateur photographers, I started with film cameras. With the advent of digital technology, digital cameras have become popular, and I have decided to purchase one. To ensure that my money is spent wisely, I need to do some homework first.

Most professional and amateur photographers use SLR (single lens reflex) cameras as they give more control to the photographer. I am familiar with film SLRs, but I have no idea about digital SLRs. So I decided to use a search engine to learn something about digital SLR cameras.

Suppose the search engine I am using has an index table like the one shown in Figure 2.7. By typing "SLR," I should get the following list, assuming w1, w2, . . . , are weights used by the engine and also, w1 > w2 > . . . , and so on:

- `www.cheapCameras.com`
- `www.buyItHere.com`
- many other links

I soon discover that pretty much all the top sites are vendor sites that sell SLR cameras. They do not discuss the performance issues of digital SLRs, which is the

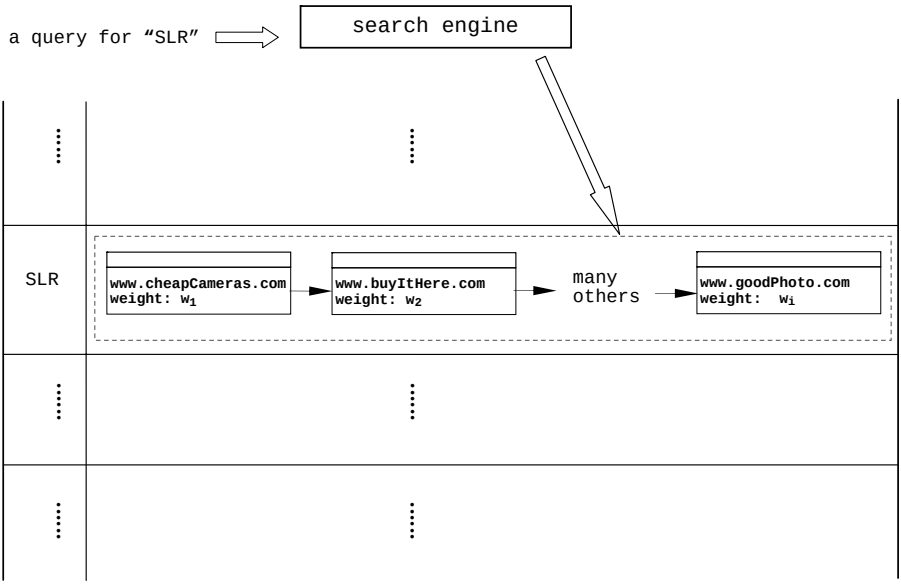

FIGURE 2.7 SLR as a keyword in the index table.

topic that I am looking for. Only if I am patient enough will I be able to reach `www.goodPhoto.com`, which is very useful and does address my concerns.

There is even more bad news. There are some really good sites I have missed. For instance, `www.digcamhelp.com` talks about shutter speed and aperture in great detail, but that particular page does not have the keyword "SLR" in it — it uses the phrase *single lens reflex* instead of SLR. For similar reasons, I missed another good site, `www.ehow.com`. This is indeed very frustrating; only after I try different keywords can I hope to see these sites in the returned list.

Let us move on to search engines in the Semantic Web environment. We can use this same hypothetical example to show how the Semantic Web can help tackle these issues.

2.2.2 Building a Semantic Web Search Engine

In the world of the Semantic Web, the search engine is modified significantly, after which a given search such as the foregoing example will return much more meaningful results. Let us examine these changes one by one. Again, the goal is not to build a Semantic Web search engine; rather, we will use this opportunity to gain a greater understanding of the Semantic Web and what it has to offer.

Step 1: Build a common language.

The word *semantics* means *meaning*. Adding semantics into the Web, therefore, means adding meaning to the Web. Before we can add meaning, the first step is to figure out how to express meaning. This is done by constructing a vocabulary that has meaning (knowledge) coded inside its terms.

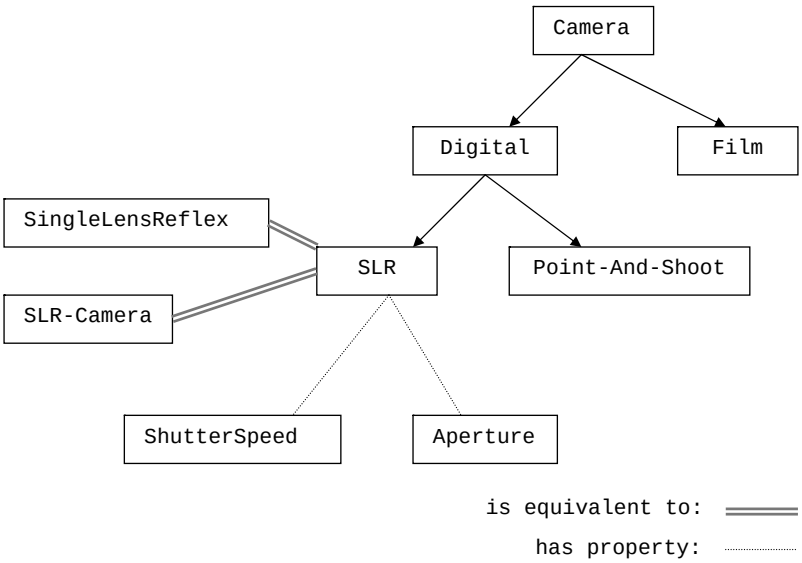

FIGURE 2.8 A small vocabulary in the domain of photography.

Let us again take the photography example. In this domain-specific area, let us assume there is a group of experts — mainly professional photographers who have extensive domain knowledge — who get together, and work out a vocabulary set as shown in Figure 2.8.

This extremely small vocabulary tells us the following:

> `Camera` is a root concept in this vocabulary; we can also call it a root term or a root class. `Camera` has two subconcepts: `Digital` camera or `Film` camera. Also, `Digital` camera can have two subclasses: `SLR` camera or `Point-and-Shoot` camera.

Note that the concept of *class* is the same as that used in the object-oriented programming language world (Java or C++ developers should feel comfortable now). At this point, we can see the meaning, or knowledge if you will, of the photography domain being coded into this vocabulary. We can even add more knowledge into it by saying more about this domain:

> `SingleLengthReflex` camera is exactly the same class or concept as the `SLR` class, and `SLR-Camera` is also exactly the same concept as the `SLR` class. Further, one can use two different concepts to describe the performance of a `SLR` class: one is `ShutterSpeed` and the other is `Aperture`; or, one can say that `SLR` class has two properties: `ShutterSpeed` and `Aperture`.

Now we are done with building the vocabulary. Clearly, it is domain specific and includes a bunch of concepts (classes) with the relations defined among these concepts. The benefits of building such a vocabulary are the following:

- It is a standard way to express the meaning/knowledge of a specific domain.
- It is a common understanding/language/meaning/knowledge shared by different parties over the Web.

Pay attention to the fact that this vocabulary is built for a specific domain, and in our example this domain is photography. Why does the vocabulary have to be domain specific? The answer is simple: building a comprehensive vocabulary that covers everything is just too ambitious; it is not even technically possible, given the fact that a single concept may have different meanings in different domains. For example, the concept of protein in the domain of biology or bioinformatics deals primarily with the structure and characteristics of the protein, and researchers in this domain are more interested in predicting the functions of a protein given its specific structure, or *vice versa*. On the other hand, in the domain of human health, "protein" may well be related to the protein content of different types of food, and the relationship between weight loss and protein consumption. It is just not possible to build a universal vocabulary to encompass all the concepts and their interrelationships.

Let us go back to the topic of building a Semantic Web search engine. Given that we have a way now to express the meanings of a given domain, what is the next step?

Step 2: Markup the pages.

As previously mentioned, one of the benefits of having a vocabulary is that different parties now have a common and shared understanding of the concepts in the domain. In our hypothetical example, let us assume that the owner of `www.goodPhoto.com` has familiarized himself with this vocabulary and that he also agrees with the meanings expressed in this vocabulary. For example, several of the pages in `www.goodPhoto.com` do contain the keyword SLR, and these pages are not concerned with the sale of SLR cameras; instead, they discuss different performance measurements of SLR cameras such as shutter speed and aperture, concepts that are expressed in the vocabulary.

Now the question is, how should the owner of `www.goodPhoto.com` explicitly indicate that the word SLR in his pages has exactly the same semantics as the concept SLR defined in the vocabulary? We will quickly see that to take advantage of the Semantic Web, semantic similarities between the words on the given page and the concepts defined in the vocabulary have to be explicitly expressed.

The solution is to *markup* the page. More specifically, to markup means to add some extra data or information to the Web page to describe some specific characteristic of the page. In our case, we need to add some data to explicitly indicate that the semantics of some words contained in this page is defined in some common vocabulary set. Now, the solution should remind us of one of the most important concepts in the Web world: metadata. The extra data used to express semantic similarity is indeed metadata. Therefore, to markup a page is to add some metadata to it.

Recall that metadata should be added to the page by following some predefined metadata schema, such as Dublin Core, which is in fact the only schema we know

now. However, Dublin Core elements seem to be fairly limited when we attempt to use them to accomplish what we want here.

Later on in this book, we will see some languages with much stronger descriptive power that have already been invented to express the semantics and knowledge for any possible domain; in fact, DC schema is absorbed by these languages. We will also see that these languages can easily accomplish what we need to do here. For now, without worrying too much about these technical details, let us just assume the markup process can be implemented by taking the two steps described in the following text.

First, the page owner of `www.goodPhoto.com` creates a special file by using one of the description languages (we will see these languages in later chapters). This file conveys the following information:

> The word SLR used in these pages has the same semantics as that defined in the common vocabulary file.

The page owner then saves this file somewhere on the Web, most likely on the Web server hosting `www.goodPhoto.com`.

Second, the pages have to be marked up: each page that has the word SLR has to be connected to the preceding description file. This is done by adding a `<link>` tag in the `<head>` section of the page, as shown in List 2.1.

LIST 2.1
Connecting the Web Page to the Special File

```
<HTML>
<HEAD>
  <TITLE>the performance of a digital camera</TITLE>
  <LINK rel="help" href=URL_of_the_special_file>
  ...other stuff
</HEAD>
...the rest of the document...
```

Note that the "`rel`" attribute has `help` as its value, meaning the link specified here refers to a document offering more information, which seems to fit our needs well. At the time of this writing, there is no standard governing the connection of a document page to its semantic markup file; using the `<link>` tag is currently an acceptable and popular approach.

The owner has now finished the markup work on the pages of `www.goodPhoto.com`. Let us now assume the owners of the following two sites have also marked up their pages:

`www.digcamhelp.com`
`www.ehow.com`

The reason why the pages on `www.digcamhelp.com` never show up in the index table in a traditional search engine is because the particular page useful to us does

not contain the word SLR, despite the fact that it gives an excellent summary of shutter speed and aperture, because of which it is still quite useful to us. Similarly, to markup this page, the owner of this site has created the special file (using this much richer language that we will come to know in later chapters) to express the following fact:

> The semantics of the words shutter speed on this page is the same as that defined in the common vocabulary; the semantics of the word aperture is the same as that defined in the common vocabulary.

The linking procedure is done similarly by the owner. Now for the pages on `www.ehow.com`; they do not show up in the returned list when using the traditional search engine because these pages use "single lens reflex" instead of "SLR." Noting that the vocabulary does include a concept "`SingleLensReflex`," the owner therefore expresses the following fact in the special file:

> The single lens reflex camera discussed on this page is an instance of the class `SingleLensReflex` defined in the common vocabulary.

The linking process is again implemented by using the `<link>` tag by the owner. What about those sites that are mainly selling the SLR cameras? There will be no markup happening on these sites at all. The reason is simple: the semantics on these sites is about selling, which is quite different from the semantics defined by the common vocabulary; therefore, the owners of those sites will never markup their pages against the common vocabulary.

Step 3: A much smarter crawler.

Now that the related pages have been marked up, it is up to the crawler to collect this information while crawling. It can now be viewed as a smart agent. Let us take a closer look at how it works.

Let us use the same example. At some point during its journey on the Web, the crawler reaches a page under `www.cheapcameras.com`. Just as it usually does, it downloads this page and starts to index the words on it. The first thing the crawler notices is that this page does not link to any special file. This implies that this page has not been marked up for any special semantics. When the crawler finally hits the word SLR, it adds the word to the index table; the document structure record now clearly shows no markup has been done on this page. The current index table is shown in Figure 2.9.

For all the pages meant mainly for selling the cameras, the same process is repeated by the crawler: when it hits the word SLR, it simply locates it in the index table, and adds the corresponding document record, as shown in Figure 2.9.

Finally, the crawler reaches a page belonging to `www.goodPhoto.com`. The first thing the crawler sees is that this page links to another file (the markup file), which specifies that the word SLR used on this page has the same semantics as that defined

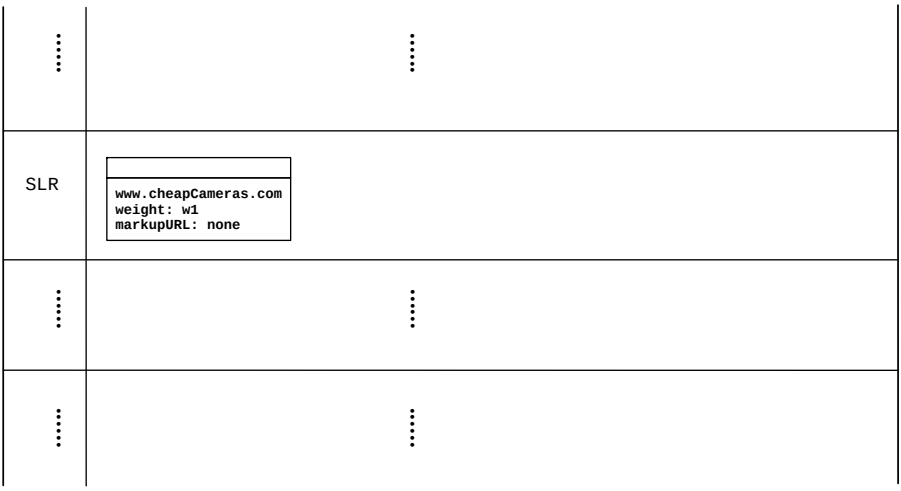

FIGURE 2.9 Index table including markup information.

in the common vocabulary. Immediately, the crawler accesses the URL of this markup file and downloads it to its memory. At this point, it will perform the following two important steps:

- Parse the markup file: After parsing the markup file, the crawler knows that the word SLR used on the current Web page has the same semantics as that defined in a common vocabulary file; let us name this common vocabulary `mySimpleCamera.owl`. You will understand why we use `owl` as part of the filename when you finish reading the later chapters.
- Download `mySimpleCamera.owl`: The crawler then downloads `mySimpleCamera.owl` into its memory for later reference.

After finishing these two steps, the crawler continues its normal work: parses the page into words and adds each word one by one into the index table. When it reaches the word SLR, it will recall that it has been described in the markup file, and its semantically equivalent concept is defined in `mySimpleCamera.owl`. The crawler then locates the word SLR in the index table and updates the table as shown in Figure 2.10.

Interesting things happen when the crawler reaches the pages on `www.digcamhelp.com`. The moment it reaches this page, the crawler will realize that this page has been marked up, and it will download the markup file. Because the markup file contains the words *shutter speed* and *aperture*, the crawler will remember them. It will also download `mySimpleCamera.owl`. After these steps, the crawler carries on with the normal processing of the page. During this process, when it hits any one of these keywords, i.e., shutter, speed, or aperture, it infers the following:

- These words have the same semantics as that defined in `mySimpleCamera.owl`.

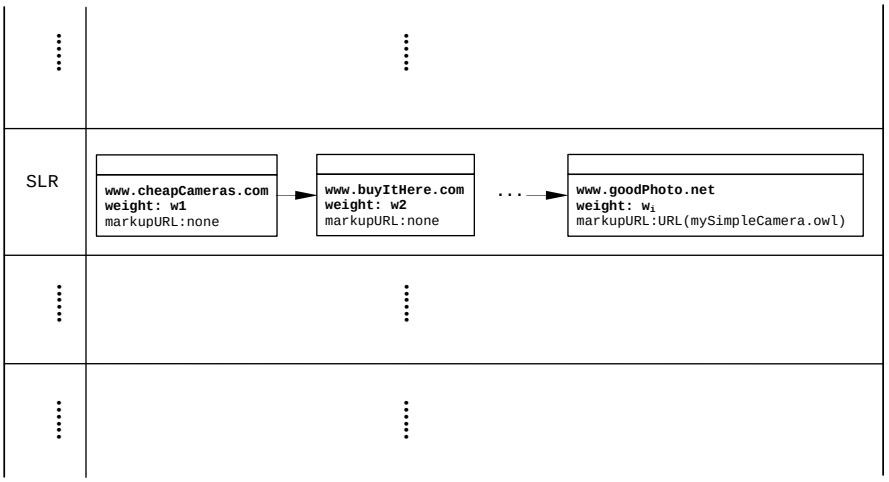

FIGURE 2.10 The index table after `www.goodPhoto.net` is added.

- In `mySimpleCamera.owl`, `ShutterSpeed` and `Aperture` are properties of the class `SLR`.
- These properties are exclusively defined to describe the properties of class `SLR`, not anything else.
- Therefore, the current document must be a document about `SLR`.

As you can tell, these are quite impressive inferences. It is also encouraging to realize that this can be easily done by using the *inference engine* associated with the description languages we have mentioned earlier. As the markup file the crawler is reading is created by using one of these description languages, the inference engine can certainly be accessed by the crawler. It is therefore easy to see why the crawler is so smart. You will understand the description languages and their inference engines when you finish reading the chapters to come.

The end result is, even though the word SLR never even appears on the page, the crawler still adds one entry into the index table under the word SLR (index key), and the added entry points to this page under `www.digcamhelp.com`. This is indeed quite impressive, and is made possible by having `mySimpleCamera.owl` defined, a markup file created and linked to the page. The current index table is shown in Figure 2.11.

It is worth emphasizing once more: using the search engine under the traditional Web, this page will never be included into the document lists corresponding to the index key `SLR`; therefore, any search based on SLR will never retrieve this page, though it is an excellent page about SLR cameras.

As the final example of the crawler's work, let us take a look at what happens when the crawler visits `www.ehow.com`. The crawler again realizes that this page has been marked up as a Semantic Web page and it therefore pays more attention to it. The result is the following inference process:

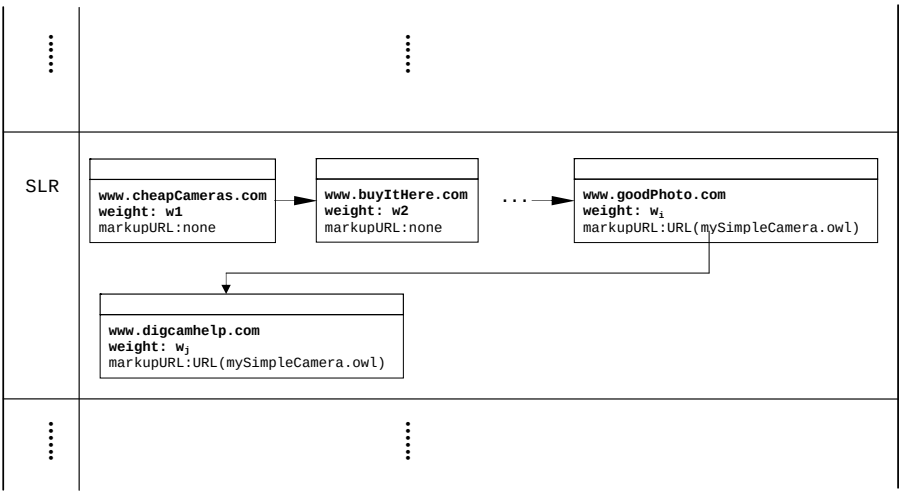

FIGURE 2.11 Page `www.digcamhelp.com` is included in the index table under SLR even though it does not contain the word SLR.

- The markup file says, "the single lens reflex camera discussed on this page is an instance of the class `SingleLensReflex` defined in `mySimpleCamera.owl`."
- Therefore, this page is about `SingleLensReflex` cameras.
- According to `mySimpleCamera.owl`, `SingleLensReflex` as a concept is equivalent to the `SLR` concept.
- Therefore, this page is also about SLR.

The crawler then adds this page as a document entry into the index table under the index key `SLR`, even though this page does not contain the word SLR. Again, as a comparison, if we are using a traditional search engine and search the Web by using SLR, we will never retrieve this page.

For our hypothetical example, we have finished discussing the crawler. Let us move on to see what happens when the real search begins: how does the search identify the right pages?

2.2.3 Using the Semantic Web Search Engine

When it comes to using our hypothetical Semantic Web search engine, you can utilize it either as an ordinary engine or as a semantic search engine. To use it as an ordinary engine, you do nothing special: enter the keyword(s) to start the search. For instance, we can still search for the keyword SLR. Now, as we did not tell the engine that we are doing a semantic search, the engine just ignores all the semantic matching power (more on this later) it has and returns the documents in which the word SLR has made an appearance.

Recall the same search we conducted using the traditional search engine. The fact is, even when this new engine is used as a traditional one, it still returns a list

that is longer than the list returned by the traditional engine. The reason has been discussed in the previous section: the crawler has added several documents that do not contain SLR as a keyword, and these documents are added as the crawler is able to make inferences based on the page markup metadata and the referenced `mySimpleCamera.owl` file. Therefore, if we use the new engine in the old way, we will have even more pages to sift through.

Now, if we would like to use this new engine as a Semantic Web search engine, we should follow the following steps.

Step 1: Select your semantics. After accepting the keyword we entered (in this case, SLR), the engine will first scan all the available common vocabularies and find all the ones that have the concept SLR defined in them. You can imagine that we are not the only ones interested in building Semantic Web search engines, so it is quite possible that there are other common vocabulary sets. The engine then presents these `owl` files, and it is up to us to tell the engine that the semantics (meaning) of the keyword SLR is the same as that defined in this vocabulary: `mySimpleCamera.owl`.

Step 2: Search the index table and return the results. The engine then goes into the index table to find the candidate pages. It does this by retrieving the document list indexed by the keyword `SLR`. It then iterates over this list and for each document in the list, it checks the `markupURL` field and does the following:

1. If this field points to `NULL`, discard this document. This field will point to a valid URL if the document is indeed marked up as a Semantic Web page. If this field points to `NULL`, then no semantics has ever been added to this page.
2. If the field is not `NULL` and if it does not point to `mySimpleCamera.owl`, then discard this document. Clearly, it could be true that the word SLR is also defined in some other common vocabulary and some owners have marked up their pages to indicate the semantics of the word SLR contained in their documents to be the same as that defined in that particular vocabulary. For us, this simply indicates that the same word can have different meanings. For the search engine, this simply means, "sorry, this page has the keyword you entered, but its meaning is different from what you are looking for, as the semantics of the word in this document is defined in a different vocabulary."
3. If the `markupURL` field is not `NULL` and it does point to `mySimpleCamera.owl`, then include this document in the candidate set. This part is obvious. For our hypothetical example, the following three sites will be collected (note that all the sites selling cameras are not included this time!):

   - `www.goodPhoto.com`
   - `www.digcamhelp.com`
   - `www.ehow.com`

Once the engine finishes scanning the index table, it has a candidate set (as described earlier). Now, the candidate set is sorted by using the weighting schema; in other words, the engine now tries to sort the documents in the set and list the most relevant ones on the top. Once this step is finished, the result is returned to the user.

This time, it does get better: all the camera-selling sites are gone, and the results are all related to what we are looking for.

2.3 FURTHER CONSIDERATIONS

You have seen one example of a Semantic Web search engine; this is just one possible way of constructing it, and the purpose is to show you how the Semantic Web can help a search engine achieve better results. It is not meant to offer a practical solution to the challenge of building a Semantic Web search engine. Building a real Semantic Web search engine will require a lot more work, and the following text describes some issues that need to be considered when building such an engine.

2.3.1 Web Page Markup Problem

One obvious issue is the markup of the Web pages. In our hypothetical example, we assume the markup work is done by each individual site owner. In reality, however, the site owner may not be motivated to do so. For instance, he or she could be waiting for a killer smart agent application to show up. The trouble is, the killer application depends on the markup of the Web page. This seems to be another example of the chicken-and-egg problem: without this killer application, no one will be motivated enough to implement the markup; however, without the markup, the killer application is simply not possible. How to solve this problem? Hopefully, W3C can find the resources to implement an application to convince the public.

Another solution to handle this is to do automatic markup by running a smart crawler — this crawler is different from the search engine crawler in that its task is not to build an index table but to automatically markup the Web pages it has visited. At the current stage, this still seems quite hard to accomplish given that Web pages are not machine readable.

2.3.2 "Common Vocabulary" Problem

When creating the example search engine, we saw the importance of the "common vocabulary" (`mySimpleCamera.owl` in our example). It is where all the semantics and knowledge are recorded for use by the crawler later on. But how to create these files? Also, we have seen that these files are normally domain specific; therefore, how many domains should we have? Who is qualified to be a domain expert?

There is more bad news. One of the difficulties is that for a single domain, there could exist several of these files, each of which tries to capture the common terminologies and their relations in the given domain. Then we have the problem of overlapping semantics. How to match a given vocabulary to another in order to decide if they are equivalent? Under what circumstances can we interchange two different concepts, each defined in separate but equivalent vocabularies? These questions have to be solved before a real Semantic Web search engine can be constructed.

2.3.3 Query-Building Problem

Another important issue is how to build the query. In our simple example, we assumed the search engine would present the user the vocabulary files and the user would decide which file defines the semantics that he or she prefers. How good is this solution? What if the user is not quite clear about the semantics of the concept he or she is searching for? What if the concept shows up in several definition files and each of them looks fairly close?

You see all these issues, and they are not all. Among the ones we have not mentioned, there are the issues of performance and scalability, which are always concerns to us. For instance, from the example we see that quite often the crawler needs to do some inference work about the facts it is collecting; this is time consuming and the performance and scalability issues cannot be ignored.

We now return to our topic. Again, a Semantic Web search engine is presented here just so that you can understand more about the Semantic Web, and how it can help us. The purpose is not to make a search engine expert out of you, at least not at this point.

So, do you have a better understanding of the Semantic Web now?

2.4 THE SEMANTIC WEB: A SUMMARY

In this chapter, two search engines were described for comparison purposes. One of these engines was constructed under the traditional Web environment and the other under the Semantic Web environment. The goal of the exercise was to gain a better understanding of the Semantic Web by understanding how the Semantic Web can help search engines deliver better search results. Let us briefly summarize what we have learned:

- In the traditional Web environment, each Web page only provides information for computers to display the page, not to understand it; the page is solely intended for human eyes.
- Therefore, traditional search engines are forced to do keyword matching only. Without any semantics embedded in the page, the user certainly gets quite a lot of irrelevant results.

To solve this problem, we can extend the traditional Web by adding semantic data to it. Here is how we do it:

- We can construct a vocabulary set that contains (all) the important words (concepts, classes) for a given domain, and the semantics and knowledge are coded into this set; more importantly, this set has to be constructed using some structured data.
- We then markup a Web page by adding a *pointer* in its metadata section. This pointer points to the appropriate vocabulary set; this is how we add semantics to each page.

- When visiting a Web page, a smart agent (or crawler in the search engine example) is able to see the link from its metadata section and follow it to retrieve the vocabulary set.
- As this set is constructed using structured data, the agent is able to understand the vocabulary set. Also, as the given page is linked to this set, the agent is able to understand what this page is all about.

Now, based on this understanding, we can frame a better definition of the Semantic Web:

> The Semantic Web is an extension of the current Web. It is constructed by linking current Web pages to a structured data set that indicates the semantics of this linked page. A smart agent, which is able to understand this structure data set, will then be able to conduct intelligent actions and make educated decisions on a global scale.

2.5 WHAT IS THE KEY TO SEMANTIC WEB IMPLEMENTATION?

Based on our latest definition of the Semantic Web, and also based on our experience with search engines in both environments, we realize that the key to the implementation of the Semantic Web is this structured data set. Once again, let us emphasize this fact: a structured data set implies that it is machine readable. In our previous engine examples, we called it the common vocabulary file.

How to construct such a structured data set? How to markup a given page using this data set? To answer these and other related questions, we need to get into the details of Semantic Web technology. It is the topic of Part 2 of this book.

A sneak preview here: a commonly accepted name for this common vocabulary file is *ontology*, and RDF, RDFS, and OWL are all description languages that you can use to construct such an ontology.

Part 2

The Nuts and Bolts of Semantic Web Technology

In Part 1 we have spent quite some time discussing what the Semantic Web is, and we have examined two search engines in great detail in order to gain a better and more concrete understanding of the Semantic Web and the value it can add. We hope this goal is well accomplished, and you are now motivated and well prepared to dive into the technical world of the Semantic Web.

The chapters in this part will examine the nuts and bolts under the surface of the Semantic Web. We will start with RDF and RDFS, proceed to ontologies and ontology languages such as OWL, and also discuss some available tools you can use as a Semantic Web developer. We will use many examples to demystify these topics. After finishing these chapters, you should be well equipped to begin a much more interesting and challenging journey into the world of the Semantic Web on your own.