

# *Intelligent web crawling*



## ***This chapter covers***

- A brief overview of web crawling and intelligent crawling
- A step-by-step implementation of a web crawler
- Crawling with Nutch
- Scalable web crawling

No one knows the exact number of web pages on the Internet. But we do know that the World Wide Web is

- Huge, with billions of web pages
- Dynamic, with pages being constantly added, removed, or updated
- Growing rapidly

Given the huge amount of information available on the Internet, how does one find information of interest?

In this chapter, we continue our theme of gathering information from outside one's application. You'll be introduced to the field of intelligent web crawling to retrieve relevant information. Search engines crawl the web periodically to index available content. You may be interested in crawling the web to harvest information

from external sites, which can then be used in your application. Search engines such as Google and Yahoo! constantly crawl the web to gather data for their search results.

**HOW BIG IS  
THE WEB?**

In late July 2008, Google announced that they had detected more than a trillion unique URLs on the web; with the internet growing by several billion individual pages every day. Of course, not all the content has been indexed by Google, but a large portion has. To get a sense of the number of pages indexed by Google it is useful to look at the number of pages indexed by Google for a site—type `site:website`, for example, `site:facebook.com`, to search for the pages indexed by Google for Facebook (this number incidentally was more than 76 million pages as of July 2008). Other providers, such as Alexa, Compete.com, and Quantcast also provide useful data on the kinds of searches carried out on various sites.

This chapter is organized in three sections:

- First, we look at the field of web crawling, how it can be used in your application, the details of the crawling process, how the process can be made intelligent, how to access pages that aren't retrievable using traditional methods, and the available public domain crawlers that you can use.
- Second, to understand the basics of intelligent (focused) crawling, we implement a simple web crawler that highlights the key concepts related to web crawling.
- Third, we use Apache Nutch, an open source Java-based scalable crawler. We also discuss making Nutch distributed and scalable, using concepts known as Hadoop and MapReduce.

## **6.1 Introducing web crawling**

Web crawling is the automated process of visiting web pages with the aim of retrieving content. The content being extracted could be in many forms: text, images, or videos. A *web crawler* is a program that systematically visits web pages, retrieves content, extracts URLs to other relevant links, and then in turn visits those links if allowed. Web crawlers are also commonly known as *web spiders*, *bots*, or *automated indexers*.

As we'll see later in this chapter, it's easy to write a simple web crawler. But building a crawler that's sophisticated enough to efficiently crawl the complete web (or parts of it that can be crawled) is a whole different ball game. We discuss these challenges throughout this chapter.

In this section, we look at how crawling the web may be useful to you, the basics of web crawling, how crawling can be made focused or intelligent, how invisible content can be made available, and some of the available open source web crawlers.

### **6.1.1 Why crawl the Web?**

The primary goal of web crawling is to collect data from external sites. Following are some of the many ways web crawling is typically used:

- *Content aggregation and indexing external content*—Search engines build a catalog of content available on the web by periodically crawling the web. They then allow their users to search for relevant content using the data retrieved. You can also aggregate external data that may be relevant to your application and present hyperlinks to those pages from within your application.<sup>1</sup>
- *Searching for specific information*—Focused crawling deals with crawling the web specifically looking for relevant information. In essence, the crawler visits a fraction of the total pages available by visiting pages that show promise. We also refer to this as *intelligent crawling*.
- *Triggering events*—Based on your domain, it may be helpful to crawl a set of web sites in search of relevant information that can be used as triggers or events in your application. For example, if your application is in the real estate domain, where you provide a valuation for a house, it may be helpful to crawl public domain sites that post information about recent sales. A house sale could be an event in your financial model for the price of homes in a neighborhood.
- *Detecting broken links*—Since a web crawler is good at visiting a page, extracting hyperlinks, and then visiting those pages, you can use a web crawler to detect whether you have any broken links in your application.
- *Searching for copyright infringement*—If you have copyrighted content, you can use intelligent web crawling to detect sites that are inappropriately using your content.

Next, let's look at the process of crawling the web used by web crawlers.

### 6.1.2 The crawling process

The basic process of web crawling is fairly straightforward, as shown in figure 6.1. Later, in section 6.2, we implement a simple web crawler using the algorithm outlined in figure 6.1. The basic steps involved in web crawling are

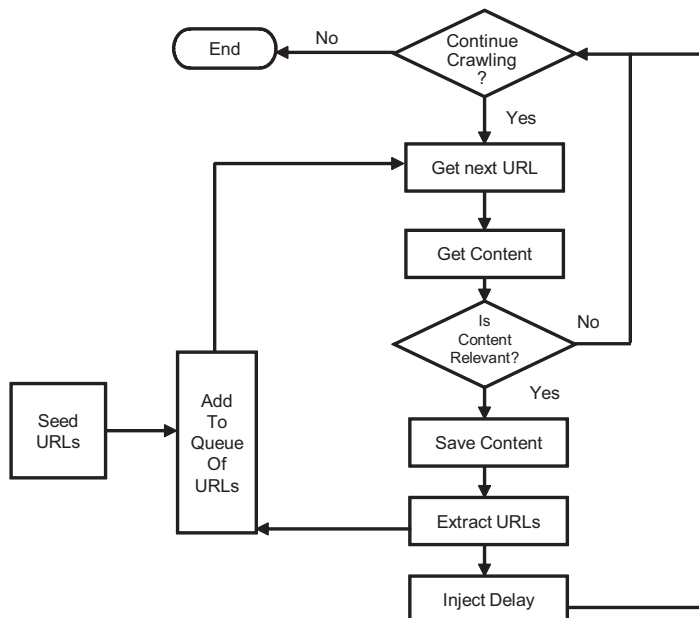
- 1 *Seeding the web crawler*—The web crawler is seeded with a set of URLs to be visited. Place these URLs in a queue data structure. The crawler queries the queue to get the next URL that it needs to crawl.
- 2 *Checking for exit criteria*—As long as the criteria for the crawler to continue crawling are met, the crawler retrieves the next URL to be visited. The exit criteria for a crawler could be based on one of many conditions—when the number of pages retrieved or visited reaches a certain threshold, how long the crawler has run, when there are no more available URLs to visit, and so on.
- 3 *Get the next URL to be visited*—There are some sites that don't allow crawlers to visit certain pages. Sites typically give permissions to crawlers to visit a set of pages in a file called robots.txt. The next step is to get the next URL that the crawler is allowed to visit.

---

<sup>1</sup> It may not be appropriate to show the content within your application if it is copyrighted.

- 4 *Retrieve content*—The crawler visits the URL and retrieves the content. Since you want the crawling process to be efficient—to avoid duplication—the URL visited is added to the list of URLs visited.
- 5 *Is the retrieved content relevant?*—This step is optional and is used when implementing focused crawling. Here, the retrieved content is checked against a model to see if the content is relevant to what we’re searching for. If it is, then the content is saved locally.
- 6 *Extract URLs*—When the content is of interest, the text in the content is parsed for hyperlinks. Hyperlinks that haven’t been visited are then placed in the queue of URLs.
- 7 *Inject delay*—If the crawling process is too fast or if multiple threads are being used, it can sometimes overwhelm the site. Sites protect themselves by blocking the IP addresses of misbehaving crawlers. You may want to optionally inject a delay between subsequent hits to a site.

This algorithm carries out its search using *breadth-first search (BFS)*—the roots of extracted URLs are visited first before going deep and visiting the children. Given that there are costs associated with the time to crawl, the network bandwidth used, and the number of machines used in the crawl, sophisticated methods have been developed to determine which URL should be visited next. These methods to estimate the potential quality of a URL include using historic information from previous crawls to determine a URL’s weight, using the number of pages connecting to the URL (also known as *authority*) and the number of outward links, and analyzing the graph of connections on the site. Of course, though understanding these basic analysis algorithms is conceptually simple, the practical details of implementing them are complex and usually proprietary.



**Figure 6.1** The basic process of web crawling

Web crawlers typically work in combination with a search library, which is used to index and search the content retrieved. For example, Nutch, which we use in section 6.3, uses Lucene, a Java-based open source search engine library, which we also use later in this book.

Given the dynamic nature of the Web, with pages being constantly added, modified, or deleted, crawling is performed periodically to keep the pages fresh. A site may have a number of *mirror* sites, and smart crawlers can detect these sites and avoid duplication by downloading from the fastest or freshest server.

A crawler can face a number of challenges during the crawl process, one of them being a *spider trap*. A spider trap could be created unintentionally or intentionally to guard a site against spam crawlers. Common techniques used to create spider traps include creating an infinitely deep directory structure and creating an infinite number of dynamically generated pages. Most crawlers stay with five levels of URL hierarchy.

Spammers use a variety of methods to mislead crawlers and boost their search engine rankings. These techniques include

- *Doorway pages*—Pages that are optimized for a single keyword, which then redirects to the real target page.
- *Keyword spamming*—Location and word frequency are two commonly used metrics used by text analysis algorithms. Spammers add misleading meta-keywords, repeat words excessively, or add hidden text.
- *Link spamming*—Some sites have numerous domains that point or redirect to a target page.
- *Cloaking*—Some sites detect when a request is from a web crawler and may serve fake spam content.

With this general overview of the crawling process, let's next focus on step 5 of the crawling process—focused crawling, where we want to focus the crawling process to get only items of interest.

### 6.1.3 Intelligent crawling and focused crawling

Given the sheer size of the web and the time and cost associated, crawling the complete Web can be daunting and potentially infeasible. Many times you may be interested in gathering information relevant to a particular domain or a topic; this is where focused crawling, also known as *topical crawling*, comes into play. Focused crawling is based on the simple principle that the more relevant a page to a topic of interest, the higher the probability that the linked pages contain relevant content. Therefore, it's advantageous to first explore these linked pages. A simple way to compute the relevancy of a page to a topic of interest is to match on keywords. The use of similarity computation between two term vectors using the term-frequency and inverse-document-frequency (TF-IDF) computation is a generalization of this idea.

Charkrabarti formally introduced focused crawling in 1999. In focused crawling, one first builds a model, also known as a *classifier*, that's trained to predict how relevant a piece of content is to the topic being searched. If you're searching for content

that's similar to a set of documents, a simple approach is to create a composite term-vector representation, similar to `CompositeContentTypes`, which we looked at in section 4.4.1, and then compute the similarity between the retrieved content and this composite representation. Of course, there are a variety of approaches that can be used to build predictive models, which we discuss in part 2 of this book.

Assume that we've built a classifier that can emit a number between 0 and 1 to predict the *relevancy* of a piece of content, such that the higher the number, the higher the probability that the item is relevant to our topic. Content that has a value above a certain threshold is accepted, and hyperlinks from these pages are added to the pool of URLs to be visited with the weight of the relevancy of the parent. The URLs in the URL queue are sorted by relevancy, such that the URL with the highest predicted relevancy is selected.

A metric commonly used to measure the effectiveness of a crawler is its *harvest rate*: the proportion of pages retrieved that are relevant to the topic of interest for the crawler. Typically, a number of heuristics depending on the domain and the item being searched for are used to improve a crawler's harvest rate.

In the sixth step of the crawling process, we discussed how the crawler finds additional links by parsing through the content. However, not all content is accessible through this process. Next, let's look at how this content can be made available to the web crawler.

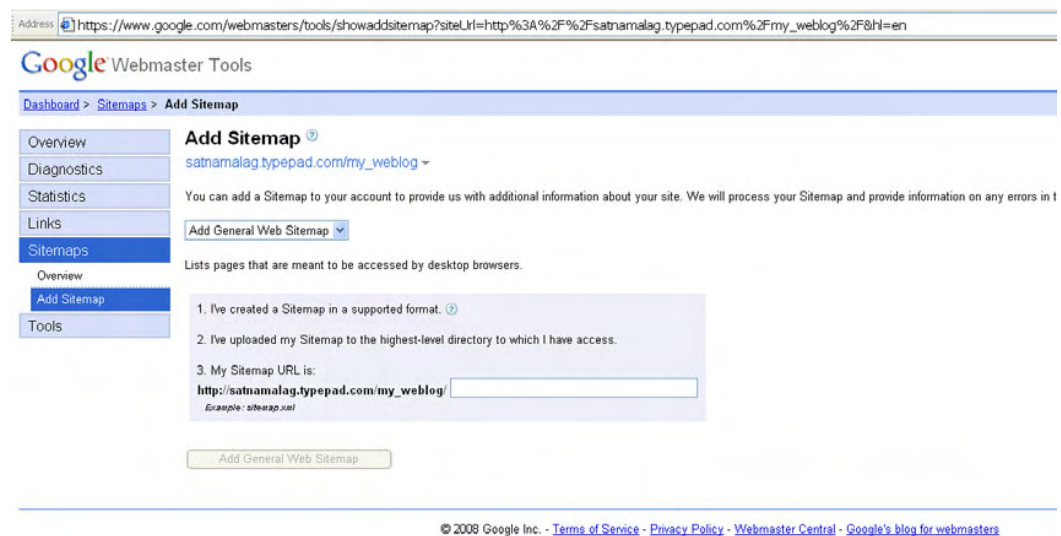
#### **6.1.4 Deep crawling**

Deep or invisible web pages are pages that can't be reached by following the links on a page. This is especially true when a site uses AJAX and crawlers can't navigate to the content. One way to solve this is to use the sitemaps protocol, a URL inclusion protocol. Sitemaps work together with the robots.txt specification, which is a URL exclusion protocol. The sitemap protocol has wide adoption, including support from AOL, Microsoft, and Yahoo!

Sitemaps are XML documents that tell web crawlers which URLs are available for crawling. They also contain additional information about the URLs, such as how often they change, when they were last updated, and the relative importance of a URL with respect to other URLs on the site. For more details on the sitemaps specification, refer to the official site at <http://www.sitemaps.org/protocol.php> and the Google sitemap site at <https://www.google.com/webmasters/tools/docs/en/protocol.html>. Since there is a limit of 50,000 URLs and up to 10MB for a sitemap file, you can also use a sitemap index specifying the locations of your sitemap files. You can have up to 1,000 sitemaps, and can specify the location of your sitemap in your robots.txt file, which we look at in section 6.2.2.

Leveraging search engine optimization and the use of sitemaps is one of the cheapest ways of marketing your content. When done correctly, your application web page will show up high in the search results of search engines such as Google and Yahoo!, and this could generate relevant traffic to your site. To show up high on results from search engines such as Google, you also need to increase the authority—sites linking to

your site—of your domain. For more information on making your site Google crawler-friendly, check out the Google Webmaster site: <https://www.google.com/webmasters/tools/docs/en/about.html>. Using the Google Webmaster tools, you can see information about what content has been indexed by the Google crawler, Googlebot; when it was last indexed; pages that the crawler had problems with; and so forth. As shown in figure 6.2, you can also submit a link to your sitemap file through the Google Webmaster tools and check for any errors in the XML files.



**Figure 6.2** Submitting your site's sitemap using Google Webmaster tools

It's also helpful to list your site with the Open Directory Project (<http://www.dmoz.org/>), which is used by a number of search crawlers and will help you increase the page rank for your site.

### 6.1.5 Available crawlers

A number of open source web crawlers written in Java are available. Refer to <http://www.manageability.org/blog/stuff/open-source-web-crawlers-java/view> for a list of available open source crawlers, with Nutch<sup>2</sup> and Heritrix<sup>3</sup> perhaps being the two most popular.

Heritrix is the Internet Archive's open source, extensible, web-scale web crawler project. Nutch was built by Doug Cutting in an effort to provide a free and open web search engine. It's built on top of Lucene,<sup>4</sup> which we use in chapter 11 when we discuss intelligent search, and has shown good scalability. Nutch has been designed to scale to over 1 billion pages, and a demo index of more than 100 million documents was created in 2003. Both Nutch and Lucene are Apache projects and carry the Apache license.

<sup>2</sup> <http://lucene.apache.org/nutch/>

<sup>3</sup> <http://crawler.archive.org/>

<sup>4</sup> Lucene is an open source software library for full-text search.



Later, in section 6.3, we use Nutch to crawl the web. Before we use an out-of-the-box crawler such as Nutch, it's useful to go through the process of building a web crawler ourselves—it'll help us better appreciate the complexities associated with crawling, especially as we try to make it scale.

Now that we know the basics of web crawling, let's systematically build a simple web crawler. This will give us useful insight into the inner workings of web crawlers and some of the issues related to web crawling.

## 6.2 *Building an intelligent crawler step by step*

In this section, we build a simple focused web crawler that follows hyperlinks to gather URLs of interest. To make the crawl focused, we use a regular expression matcher as the model for computing the relevance of content visited by the crawler. I've found this crawler to be useful in retrieving content of interest when I'm researching a particular topic. In our example, we retrieve content related to “collective intelligence” and seed the crawler by pointing it to the page on Wikipedia on collective intelligence.

### 6.2.1 *Implementing the core algorithm*

To implement our crawler, we build two classes:

- 1 NaiveCrawler implements the crawling process.
- 2 CrawlerUrl encapsulates the URL visited by the crawler.

Let's begin by looking at listing 6.1, which shows the `crawl()` method implemented by the crawler. This method follows the steps outlined in figure 6.1.

**Listing 6.1** The `crawl()` method in the `NaiveCrawler` class

```
public void crawl() throws Exception {
    while (continueCrawling()) {           ❶
        CrawlerUrl url = getNextUrl();     ❷
        if (url != null) {
            printCrawlInfo();
            String content = getContent(url); ❸
            if (isContentRelevant(content, regexSearchPattern)) { ❹
                saveContent(url, content);    ❺
                Collection<String> urlStrings =
                    extractUrls(content, url); ❻
                addUrlsToUrlQueue(url, urlStrings); ❼
            } else {
                System.out.println(url + " is not relevant ignoring ...");
            }
            Thread.sleep(this.delayBetweenUrls); ❽
        }
    }
    closeOutputStream();
}
```

The crawler consists of eight steps:

- ❶ `continueCrawling()`: checks to make sure that the crawl exit criteria aren't met
- ❷ `getNextUrl()`: gets the next URL to be visited



- ③ `getContent(url)`: retrieves the content associated with the URL
- ④ `isContentRelevant(content, this.regexpSearchPattern)`: checks to see if the retrieved content is of interest
- ⑤ `saveContent(url, content)`: saves the content if the URL is of interest
- ⑥ `extractUrls(content, url)`: extracts URLs by parsing the content
- ⑦ `addUrlsToUrlQueue(url, urlStrings)`: adds the extracted URLs to the URL queue
- ⑧ `Thread.sleep(this.delayBetweenUrls)`: injects a delay before processing the next URL

We visit each of these steps in the next few sections. Before we go too far, let's look at the constructor for the `NaiveCrawler`, which is shown in listing 6.2.

### Listing 6.2 The constructor for `NaiveCrawler`

```
package com.alag.ci.webcrawler;

import java.io.*;
import java.net.URL;
import java.util.*;
import java.util.regex.*;

import org.apache.commons.httpclient.*;
import org.apache.commons.httpclient.methods.GetMethod;
import org.apache.commons.httpclient.params.HttpMethodParams;

public class NaiveCrawler {
    private static final String USER_AGENT = "User-agent:";
    private static final String DISALLOW = "Disallow:";
    public static final String REGEXP_HTTP = "<a href=\"http://(.)*\">";
    public static final String REGEXP_RELATIVE = "<a href=\"(.)*\">";
    private int maxNumberUrls;
    private long delayBetweenUrls;
    private int maxDepth;
    private Pattern regexpSearchPattern;
    private Pattern httpRegex;
    private Pattern relativeRegex;
    private Map<String, CrawlerUrl> visitedUrls = null;
    private Map<String, Collection<String>> sitePermissions = null;
    private Queue<CrawlerUrl> urlQueue = null;
    private BufferedWriter crawlOutput = null;
    private BufferedWriter crawlStatistics = null;
    private int numberItemsSaved = 0;

    public NaiveCrawler(Queue<CrawlerUrl> urlQueue, int maxNumberUrls,
        int maxDepth, long delayBetweenUrls, String regexpSearchPattern)
        throws Exception {
        this.urlQueue =
urlQueue;
        this.maxNumberUrls = maxNumberUrls;
        this.delayBetweenUrls = delayBetweenUrls;
        this.maxDepth = maxDepth;
        this.regexpSearchPattern = Pattern.compile(regexpSearchPattern);
        this.visitedUrls = new HashMap<String, CrawlerUrl>();
    }
}
```

**Regular expression to extract out URLs**

**Regular expression pattern to focus crawling**

**Map keeps track of URLs visited**

**Map keeps track of site permissions**

**Extract next URL to visit**

**Output URLs stored in two files**

```

        this.sitePermissions = new HashMap<String, Collection<String>>();
        this.httpRegexp = Pattern.compile(REGEXP_HTTP);
        this.relativeRegexp = Pattern.compile(REGEXP_RELATIVE);
        crawlOutput = new BufferedWriter(new FileWriter("crawl.txt"));
        crawlStatistics = new BufferedWriter(new FileWriter(
            "crawlStatistics.txt"));
    }

```

The crawler is seeded with an initial queue of `CrawlerUrls`, the maximum number of URLs to be visited, the maximum depth to be visited, the delay to be injected between visiting URLs, and a regular expression to guide the crawling process. This is shown in the constructor:

```

public NaiveCrawler(Queue<CrawlerUrl> urlQueue, int maxNumberUrls,
    int maxDepth, long delayBetweenUrls, String regexpSearchPattern)
    throws Exception

```

At this point it's also helpful to look at the code for `CrawlerUrl`, as shown in listing 6.3.

### Listing 6.3 The code for `CrawlerUrl`

```

package com.alag.ci.webcrawler;

import java.net.MalformedURLException;
import java.net.URL;

public class CrawlerUrl {
    private int depth = 0;
    private String urlString = null;
    private URL url = null;
    private boolean isAllowedToVisit;
    private boolean isCheckedForPermission = false;
    private boolean isVisited = false;

    public CrawlerUrl(String urlString, int depth) {
        this.depth = depth;
        this.urlString = urlString;
        computeURL();
    }

    private void computeURL() {
        try {
            url = new URL(urlString);
        } catch (MalformedURLException e) {
            // something is wrong
        }
    }

    public URL getURL() {
        return this.url;
    }

    public int getDepth() {
        return this.depth;
    }

    public boolean isAllowedToVisit() {

```

**Depth of URL**

**String value for URL**

**Determines if crawler is allowed to visit this URL**

**Determines if crawler has visited this URL**

```

        return isAllowedToVisit;
    }

    public void setAllowedToVisit(boolean isAllowedToVisit) {
        this.isAllowedToVisit = isAllowedToVisit;
        this.isCheckedForPermission = true;
    }

    public boolean isCheckedForPermission() {
        return isCheckedForPermission;
    }

    public boolean isVisited() {
        return isVisited;
    }

    public void setIsVisited() {
        this.isVisited = true;
    }

    public String getUrlString() {
        return this.urlString;
    }

    @Override
    public String toString() {
        return this.urlString + " [depth=" + depth + " visit="
            + this.isAllowedToVisit + " check="
            + this.isCheckedForPermission + "]";
    }

    @Override
    public int hashCode() {
        return this.urlString.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        return obj.hashCode() == this.hashCode();
    }
}

```

The `CrawlerUrl` class represents an instance of the URL that's visited by the crawler. It has utility methods to mark whether the URL has been visited and whether the site has given permission to crawl the URL.

Next, let's look in more detail at how the crawler gets the next URL for the crawl, which is shown in listing 6.4.

#### Listing 6.4 Getting the next url for the crawler

```

private boolean continueCrawling() {
    return ((!urlQueue.isEmpty()) && (getNumberOfUrlsVisited() <
        this.maxNumberUrls));
}

private CrawlerUrl getNextUrl() {
    CrawlerUrl nextUrl = null;
}

```

Next unvisited URL in queue

Run till out of URLs or exceed threshold

```

while ((nextUrl == null) && (!urlQueue.isEmpty())) {
    CrawlerUrl crawlerUrl = this.urlQueue.remove();
    if (doWeHavePermissionToVisit(crawlerUrl)
        && (!isUrlAlreadyVisited(crawlerUrl))
        && isDepthAcceptable(crawlerUrl)) {
        nextUrl = crawlerUrl;
    }
}
return nextUrl;
}

private void printCrawlInfo() throws Exception {
    StringBuilder sb = new StringBuilder();
    sb.append("Queue length = ").append(this.urlQueue.size()).append(
        " visited urls=").append(getNumberOfUrlsVisited()).append(
        " site permissions=").append(this.sitePermissions.size());
    crawlStatistics.append(" " + getNumberOfUrlsVisited()).append(
        ", " + numberItemsSaved).append(", " + this.urlQueue.size())
        .append(", " + this.sitePermissions.size() + "\n");
    crawlStatistics.flush();
    System.out.println(sb.toString());
}

private int getNumberOfUrlsVisited() {
    return this.visitedUrls.size();
}

private void closeOutputStream() throws Exception {
    crawlOutput.flush();
    crawlOutput.close();
    crawlStatistics.flush();
    crawlStatistics.close();
}

private boolean isDepthAcceptable(CrawlerUrl crawlerUrl) {
    return crawlerUrl.getDepth() <= this.maxDepth;
}

private boolean isUrlAlreadyVisited(CrawlerUrl crawlerUrl) {
    if ((crawlerUrl.isVisited())
        || (this.visitedUrls.containsKey(
            crawlerUrl.getUrlString()))) {
        return true;
    }
    return false;
}

```

Prints details of crawler run

Returns number of URLs visited

Closes all output streams

Checks depth of URL

Checks if URL has been visited

The method `continueCrawling` checks whether the crawler should stop crawling. Crawling stops when we run out of URLs to crawl or we've visited the maximum number of specified URLs. The method `getNextUrl` retrieves the next available URL that hasn't been visited, has acceptable depth, and that we're allowed to visit. To find out if we're allowed to visit a particular URL, we need to look at the `robots.txt` file.

### 6.2.2 Being polite: following the `robots.txt` file

A site can disallow web crawlers from accessing certain parts of the site by creating a `robots.txt` file and making it available via HTTP at the local URL `/robots.txt`. The

robots.txt<sup>5</sup> specification was created in 1994. There's no agency that enforces the permissions provided by the site to the web crawlers, but it's considered good manners to respect the permissions provided by the site to the crawlers. Web crawlers that don't follow the guidelines risk having their IP blocked.

Let's look at a sample robots.txt file that I've taken from the Manning web site. It's shown in listing 6.5. This will help us understand the structure and the terms used to allow and disallow access to certain URLs.

**Listing 6.5 Example robots.txt file at <http://www.manning.com/robots.txt>**

```
User-agent: *
Disallow: /_mm/
Disallow: /_notes/
Disallow: /_baks/
Disallow: /MMWIP/

User-agent: googlebot
Disallow: *.csi
```

Note the following about the permissions set in this robots.txt file:

User-agent: \* implies that this set of rules is valid for all web crawlers.

```
Disallow: /_mm/
Disallow: /_notes/
Disallow: /_baks/
Disallow: /MMWIP/
```

This specifies that no robots are allowed to visit content in any of the following directories: /\_mm/, /\_notes/, /\_baks/, /MMWIP/.

The next specification, User-agent: googlebot, is applicable for the web crawler googlebot, which is forbidden to visit any pages that end with .csi. The specification requires only one directory per Disallow: line.

If for whatever reason you don't want any crawlers visiting your site, simply create the following robots.txt file:

```
User-agent: *
Disallow: /
```

Next, let's look at listing 6.6, which contains the methods to parse through the robots.txt file at a site to see if the crawler is allowed to visit the specified URL.

**Listing 6.6 Parsing the robots.txt file to check for permissions**

```
public boolean doWeHavePermissionToVisit(CrawlerUrl crawlerUrl) {
    if (crawlerUrl == null) {
        return false;
    }
    if (!crawlerUrl.isCheckedForPermission()) {
        crawlerUrl
            .setAllowedToVisit(computePermissionForVisiting(crawlerUrl));
    }
}
```

---

<sup>5</sup> [www.robotstxt.org/wc/norobots.html](http://www.robotstxt.org/wc/norobots.html)

```

    return crawlerUrl.isAllowedToVisit();
}

private boolean computePermissionForVisiting(CrawlerUrl crawlerUrl) {
    URL url = crawlerUrl.getURL();
    boolean retValue = (url != null);
    if (retValue) {
        String host = url.getHost();
        Collection<String> disallowedPaths =
            this.sitePermissions.get(host);
        if (disallowedPaths == null) {
            disallowedPaths = parseRobotsTxtFileToGetDisallowedPaths(
                host);
        }
        String path = url.getPath();
        for (String disallowedPath : disallowedPaths) {
            if (path.contains(disallowedPath)) {
                retValue = false;
            }
        }
    }
    return retValue;
}

private Collection<String> parseRobotsTxtFileToGetDisallowedPaths(
    String host) {
    String robotFilePath = getContent(
        "http://" + host + "/robots.txt");
    Collection<String> disallowedPaths = new ArrayList<String>();
    if (robotFilePath != null) {
        Pattern p = Pattern.compile(USER_AGENT);
        String[] permissionSets = p.split(robotFilePath);
        String permissionString = "";
        for (String permission : permissionSets) {
            if (permission.trim().startsWith("*")) {
                permissionString = permission.substring(1);
            }
        }
        p = Pattern.compile(DISALLOW);
        String[] items = p.split(permissionString);
        for (String s : items) {
            disallowedPaths.add(s.trim());
        }
    }
    this.sitePermissions.put(host, disallowedPaths);
    return disallowedPaths;
}

```

**Site permissions  
cached in local  
variable**

**Parses content  
of robots.txt**

**Checks for  
disallowed path**

**Gets robots.txt file**

**Pattern  
matching  
to extract  
disallowed  
paths**

Once the robots.txt file has been parsed for a site, the permissions are cached in a local variable `disallowedPaths`. This ensures that we don't waste resources hitting the site unnecessarily.

The location of the sitemap can be specified in the robots.txt file by adding the following line (see section 6.1.4):

```
Sitemap: <sitemap location>
```

Next, let's look at how our crawler retrieves content.

### 6.2.3 Retrieving the content

As in the previous chapter, we retrieve content from a URL by using the `org.apache.commons.httpclient` package, as shown in listing 6.7.

**Listing 6.7 Retrieving content from URLs**

```
private String getContent(String urlString) {
    return getContent(new CrawlerUrl(urlString, 0));
}

private String getContent(CrawlerUrl url) {
    HttpClient client = new HttpClient();
    GetMethod method = new GetMethod(url.getUrlString());
    method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpMethodRetryHandler(3, false));
    String text = null;
    try {
        int statusCode = client.executeMethod(method);
        if (statusCode == HttpStatus.SC_OK) {
            text = readContentsFromStream(new InputStreamReader(method
                .getResponseBodyAsStream(),
                method.getResponseCharSet()));
        }
    } catch (Throwable t) {
        System.out.println(t.toString());
        t.printStackTrace();
    } finally {
        method.releaseConnection();
    }
    markUrlAsVisited(url);
    return text;
}

private static String readContentsFromStream(Reader input)
    throws IOException {
    BufferedReader bufferedReader = null;
    if (input instanceof BufferedReader) {
        bufferedReader = (BufferedReader) input;
    } else {
        bufferedReader = new BufferedReader(input);
    }
    StringBuilder sb = new StringBuilder();
    char[] buffer = new char[4 * 1024];
    int charsRead;
    while ((charsRead = bufferedReader.read(buffer)) != -1) {
        sb.append(buffer, 0, charsRead);
    }
    return sb.toString();
}

private void markUrlAsVisited(CrawlerUrl url) {
    this.visitedUrls.put(url.getUrlString(), url);
    url.setIsVisited();
}
```

**Retrieves content from URL**

**Used as recommended by package**

**Converts stream to String representation**

**Marked as visited when content is retrieved**

The method `getContent` retrieves the content for the specified URL using the `HttpClient` and `GetMethod` objects. The method extracts the content from the visited URL



by using the method `readContentsFromStream`. Once we have the content, we need to go through it to extract additional URLs that are within the content.

### 6.2.4 Extracting URLs

We extract two types of hyperlinked URLs.

First are hyperlinks with an absolute path, for example:

```
<a href=http://www.bath.ac.uk/carpp/davidskrbina/chap8.pdf
  class="external autonumber">[1]</a>
```

For this, we use the simple regular expression `<a href=\"http://(.)*\">`. This regular expression looks for strings starting with `<a href=\"http://\"` and ending with the matching `>`. Note the `\` after `href=`, which is used to escape the `"` character in Java.

Second are relative paths, an example of which is

```
<li><a href="/wiki/Systems_intelligence" title="Systems intelligence">
  Systems intelligence</a></li>
```

For this, we use the simple regular expression `<a href=\"(.)*\">`. The code to extract these two kinds of URLs is shown in listing 6.8.

#### Listing 6.8 Extracting the URLs

```
public List<String> extractUrls(String text, CrawlerUrl crawlerUrl) {
    Map<String, String> urlMap = new HashMap<String, String>();
    extractHttpUrls(urlMap, text);
    extractRelativeUrls(urlMap, text, crawlerUrl);
    return new ArrayList<String>(urlMap.keySet());
}

private void extractHttpUrls(Map<String, String> urlMap, String text) {
    Matcher m = httpRegexp.matcher(text);
    while (m.find()) {
        String url = m.group();
        String[] terms = url.split("a href=\"");
        for (String term : terms) {
            if (term.startsWith("http")) {
                int index = term.indexOf("\"");
                if (index > 0) {
                    term = term.substring(0, index);
                }
                urlMap.put(term, term);
            }
        }
    }
}

private void extractRelativeUrls(Map<String, String> urlMap,
    String text, CrawlerUrl crawlerUrl) {
    Matcher m = relativeRegexp.matcher(text);
    URL textURL = crawlerUrl.getURL();
    String host = textURL.getHost();
    while (m.find()) {
        String url = m.group();
        String[] terms = url.split("a href=\"");
```

**Extracts  
HTTP-based  
absolute URLs**

**Extracts  
relative  
URLs**

```

for (String term : terms) {
    if (term.startsWith("/")) {
        int index = term.indexOf("\"");
        if (index > 0) {
            term = term.substring(0, index);
        }
        String s = "http://" + host + term;
        urlMap.put(s, s);
    }
}
}

private void addUrlsToUrlQueue(CrawlerUrl url,
    Collection<String> urlStrings) {
    int depth = url.getDepth() + 1;
    for (String urlString : urlStrings) {
        if (!this.visitedUrls.containsKey(urlString)) {
            this.urlQueue.add(new CrawlerUrl(urlString, depth));
        }
    }
}

```

← Adds extracted URLs to queue if unvisited

The method `extractUrls` extracts two types of URLs. First, using the method `extractHttpUrls`, it extracts URLs that begin with `http` and follow a particular pattern. Second, using the method `extractRelativeUrls`, it extracts relative URLs using the `a href` prefix. All extracted URLs are added to the queue.

So far we've looked at the basic implementation of the crawler. Next, let's look at what's required to make the crawler intelligent or focused.

## 6.2.5 Making the crawler intelligent

To guide the crawling process, we use a simple regular expression pattern matcher as shown in listing 6.9.

### Listing 6.9 Checking for relevant content

```

public static boolean isContentRelevant(String content,
    Pattern regexpPattern) {
    boolean retValue = false;
    if (content != null) {
        Matcher m = regexpPattern.matcher(content.toLowerCase());
        retValue = m.find();
    }
    return retValue;
}

private void saveContent(CrawlerUrl url, String content)
    throws Exception {
    this.crawlOutput.append(url.getUrlString()).append("\n");
    numberItemsSaved++;
}

```

← Content is relevant when it matches pattern

← Relevant URLs written to file

The implementation of the `isContentRelevant` method simply tests to see if the content matches a regular expression. In our case, the `saveContent` method simply adds the URL to the list of interesting URLs.

We're now ready to make the crawler do some work for us.

### 6.2.6 Running the crawler

Now we're ready to launch our crawler to find relevant information for us in the web. Since this book is about collective intelligence, we use our crawler to find content related to collective intelligence. We seed our crawler with the page on Wikipedia relating to collective intelligence—[http://en.wikipedia.org/wiki/Collective\\_intelligence](http://en.wikipedia.org/wiki/Collective_intelligence)—as shown in listing 6.10.

#### Listing 6.10 Main program for the crawler

```
public static void main(String[] args) {
    try {
        Queue<CrawlerUrl> urlQueue = new LinkedList<CrawlerUrl>();
        String url =
            "http://en.wikipedia.org/wiki/Collective_intelligence";

        String regexp = "collective.*intelligence";
        urlQueue.add(new CrawlerUrl(url, 0));
        NaiveCrawler crawler = new NaiveCrawler(urlQueue, 2000, 5,
            1000L, regexp);
        crawler.crawl();
    } catch (Throwable t) {
        System.out.println(t.toString());
    }
}
```

Seed crawler with Wikipedia page

Visit 2000 sites, depth of 5, wait 1 second

Crawl focused for terms *collective* and *intelligence*

In our main program, we simply seed the crawler with a link to the Wikipedia site, set it to search for content having phrase *collective intelligence*, and allow the crawler to crawl.

Figure 6.3 shows the graph of the number of relevant URLs found by the crawler as a function of how many URLs it visits. Note the couple of steep gradients where the crawler finds a bunch of relevant content and then chugs along.

Listing 6.11 shows a sample of the relevant URLs that were discovered by the crawler. Imagine the usefulness of this tool when you're researching a particular topic of interest. This simple crawler can save you a considerable amount of time and effort by automating the process of following hyperlinks to discover relevant content.

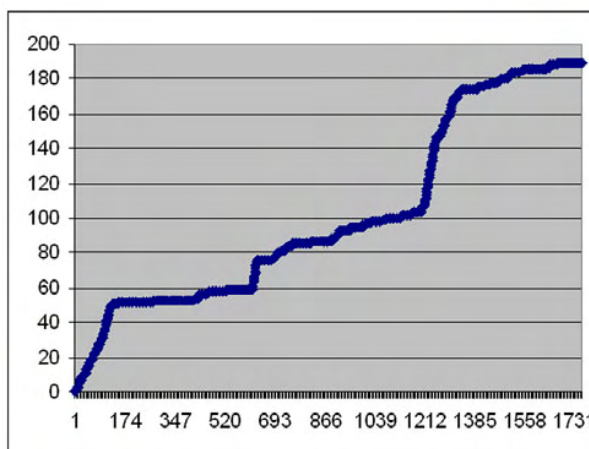


Figure 6.3 Number of relevant URLs retrieved as a function of number of URLs visited

**Listing 6.11 Sample of the URLs retrieved by the crawler**

```

http://en.wikipedia.org/wiki/Collective_intelligence
http://en.wikipedia.org/wiki/Douglas_Engelbart
http://en.wikipedia.org/wiki/Francis_Heylighen
http://ko.wikipedia.org/wiki/%EC%A7%91%EB%8B%A8%EC%A7%80%EC%84%B1
http://de.wikipedia.org/wiki/Kollektive_Intelligenz
http://en.wikipedia.org/wiki/Category:Collective_intelligence
http://en.wikipedia.org/wiki/Superorganism
http://en.wikipedia.org/wiki/Crowd_psychology
http://pt.wikipedia.org/wiki/Intelig%C3%Aancia_coletiva
http://en.wikipedia.org/wiki/Collaborative_filtering
http://en.wikipedia.org/wiki/Group_think
http://zh.wikipedia.org/wiki/%E7%BE%A4%E9%AB%94%E6%99%BA%E6%85%A7
http://www.TheTransitioner.org
http://it.wikipedia.org/wiki/Intelligenza_collettiva
http://en.wikipedia.org/wiki/Swarm_Intelligence
http://en.wikipedia.org/wiki/Special:Whatlinkshere/Collective_intelligence
http://www.axiopole.com/pdf/Managing_collective_intelligence.pdf
http://www.communicationagents.com/tom_atlee/
http://cci.mit.edu/index.html
http://www.pmcluster.com/
...

```

In section 11.5.4, we use this output to create a specialized search engine.

So far in this section, we've implemented a simple web crawler and made it intelligent. This gave you an overview of the various components that are required to make a crawler. Before we can use this in the real world, we need to make some enhancements, which we look at next.

**6.2.7 Extending the crawler**

If your goal is to crawl the entire Web or major parts of it, you'll need a crawler that's much more efficient than our simple single-threaded crawler. The bottleneck in a typical crawling process is the network delay in downloading content. Ideally, you want to download content from different hosts simultaneously. For this, you may want to enhance the crawler to have a pool of worker threads that work in parallel drawing URLs from the URL queue. Or even better, you may want to enhance the crawler to execute in parallel on distributed machines. You'll need to store the URLs visited and the queue of URLs to be visited in a database that's accessible to all the machines. You may also want to partition the URL space among the many machines, perhaps using namespaces for the URLs.

The model that we've used to focus the search is a simple pattern matcher. You'll want to use more sophisticated models, which we develop in the second part of this book. You may also want to enhance your crawler to visit URLs based on their relevance. Some crawling processes prefer to visit URLs that are referenced by many other sites; that is, sites with a high authority. Hubs—summary pages with many outgoing links—are also typically preferred by crawlers.

Given the dynamic nature of the web, you'll want to visit pages periodically to keep the content fresh. Commercial crawlers refresh content more often from sites that have shown to be historically more dynamic. For example, a news site or the home

page of a site with user-generated content is far more dynamic than static web pages for a company web site. Efficient crawlers also have a way to detect mirror sites and duplicate pages that a site may contain. Our simple crawler injects time delay between successive URL requests; you may want to enhance it to inject a delay between successive URL requests to the same host.

By this time, you should have a good understanding of how a web crawler works and the issues related to building a truly scalable web crawler, which is a nontrivial task. For large-scale web crawling, you'll really want to use a scalable open source crawler, such as Nutch, which we look at next.

### 6.3 Scalable crawling with Nutch

Nutch is a Java-based open source web crawler that has been demonstrated to scale well. It was developed by Doug Cutting and is built on top of Lucene, an API for indexing and searching that we use throughout this book. Nutch uses a plug-in-based architecture, allowing it to be easily customized. Its processing is segmented, allowing it to be distributed. Nutch consists of two main components: the crawler and the searcher.

There are some excellent freely available tutorials to help set up Nutch and crawl the Web.<sup>6</sup> There are a couple of excellent articles on Java.net by Tim White that provide a great overview of how to crawl and search with Nutch version 0.7.

In this section, we briefly go through the process of setting up and running version 0.9 of Nutch on Windows. This section consolidates information from the many tutorials and articles available on the Net, and there are some differences between 0.9 and the earlier versions. This section may well save you hours going through the various tutorials on the Web. After this section, we talk about more advanced concepts of Hadoop and MapReduce, which are used by Nutch to scale and run in a distributed mode. If you're serious about large-scale crawling, you probably want to go through this section and its referenced material in detail.

#### 6.3.1 Setting up Nutch

Let's set up Nutch to crawl Wikipedia, starting with the Wikipedia page on collective intelligence. For this, we seed the engine with the same base URL that we used in the previous section: [http://en.wikipedia.org/wiki/Collective\\_intelligence](http://en.wikipedia.org/wiki/Collective_intelligence). You need to perform the following eight steps to carry out this intranet crawl.

##### GETTING THE REQUIRED SOFTWARE

- 1 Download Nutch. You can download the latest version from <http://apache.mirrors.hoobly.com/lucene/nutch/>. This section works with version 0.9 (nutch-0.9.tar.gz; about 68Mb), which was released in April 2007. Unzip the contents of the zipped file to create a directory nutch/nutch-0.9.
- 2 Nutch requires a Unix-like environment to run its shell scripts to create indexes. If you're trying this out in the Windows environment, you'll need to download and install Cygwin (<http://www.cygwin.com/>).

---

<sup>6</sup> See [http://wiki.apache.org/nutch/Nutch\\_-\\_The\\_Java\\_Search\\_Engine](http://wiki.apache.org/nutch/Nutch_-_The_Java_Search_Engine), [http://wiki.apache.org/nutch/Nutch\\_0%2e9\\_Crawl\\_Script\\_Tutorial](http://wiki.apache.org/nutch/Nutch_0%2e9_Crawl_Script_Tutorial), and <http://lucene.apache.org/nutch/tutorial.html>.

- 3 We need a servlet container. If you don't already have one, download Apache Tomcat from <http://tomcat.apache.org/download-55.cgi>. I used Tomcat version 6.0.13 (apache-tomcat-6.0.13.zip; 6.2 Mb).
- 4 Set NUTCH\_JAVA\_HOME to the root of your JVM installation. You'll need Java 1.4.x or better. For example, I set this variable to C:\dev\Java\jdk1.5.0\_06 on my local system.
- 5 Make sure that Java, Tomcat, and Nutch are in your classpath. For example, my classpath includes C:\nutch\nutch-0.9\bin;C:\apache-tomcat-6.0.13\apache-tomcat-6.0.13\bin; %JAVA\_HOME%.

#### CREATING AN INDEX

- 6 Nutch uses a file to create an index of the content retrieved. Create a file nutch/nutch-0.9/urls on your local file system.
- 7 Create a file called seed-urls with the following as the only entry in the file:  
[http://en.wikipedia.org/wiki/Collective\\_intelligence](http://en.wikipedia.org/wiki/Collective_intelligence)

#### CONFIGURING FOR INTRANET CRAWL

- 8 Edit the file conf/crawl-urlfilter.txt and replace MY.DOMAIN.NAME with the name of the domain you wish to crawl. For our example, we limit the crawl to the wikipedia.org domain, so the line should read

```
# accept hosts in MY.DOMAIN.NAME
+^http://([a-z0-9]*\.)*wikipedia.org/
```

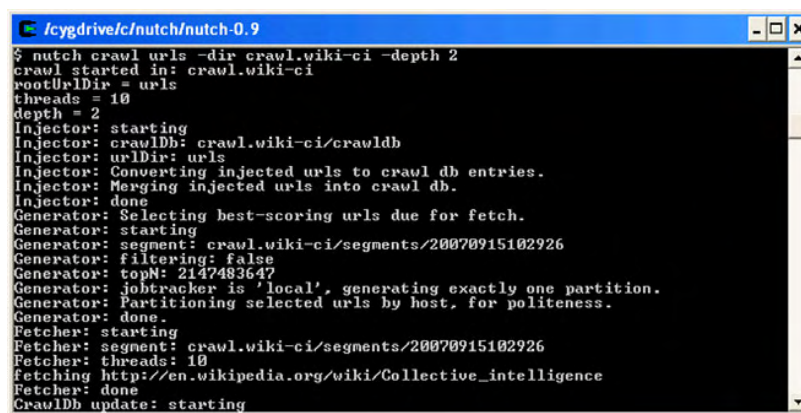
This regex will include any URL in the domain wikipedia.org.

### 6.3.2 Running the Nutch crawler

You're now ready to launch the Wikipedia crawl. In your Cygwin window, go to the directory nutch/nutch-0.9 and run this command:

```
nutch crawl urls -dir crawl.wiki-ci -depth 2
```

This launches the crawl process with a maximum depth of 2. The `-dir` option specifies the directory in which content from the crawl should be stored. This creates a directory `crawl.wiki-ci` under the nutch directory (C:\nutch\nutch-0.9\crawl.wiki-ci on my system). Your Cygwin window should be similar to the one shown in figure 6.4.



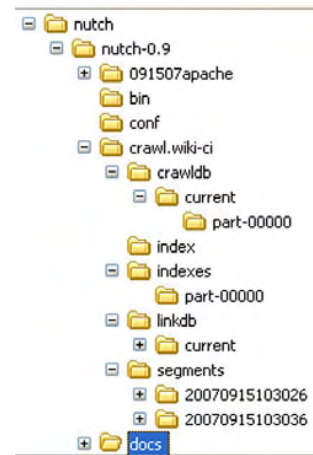
```
/cygdrive/c/nutch/nutch-0.9
$ nutch crawl urls -dir crawl.wiki-ci -depth 2
crawl started in: crawl.wiki-ci
rootUrlDir = urls
threads = 10
depth = 2
Injector: starting
Injector: crawlDb: crawl.wiki-ci/crawlDb
Injector: urlDir: urls
Injector: Converting injected urls to crawl db entries.
Injector: Merging injected urls into crawl db.
Injector: done
Generator: Selecting best-scoring urls due for fetch.
Generator: starting
Generator: segment: crawl.wiki-ci/segments/20070915102926
Generator: filtering: false
Generator: topN: 2147483647
Generator: jobtracker is 'local', generating exactly one partition.
Generator: Partitioning selected urls by host, for politeness.
Generator: done.
Fetcher: starting
Fetcher: segment: crawl.wiki-ci/segments/20070915102926
Fetcher: threads: 10
Fetching http://en.wikipedia.org/wiki/Collective_intelligence
Fetcher: done
CrawlDb update: starting
```

Figure 6.4 The Cygwin window after the crawl command

After a few minutes, the crawl should finish and there should be a new directory under `nutch-0.9` called `crawl.wiki-ci`, as shown in figure 6.5.

Nutch has created four main directories to store the crawling information:

- *CrawlDb*—Stores the state of the URLs along with how long it took to fetch, index, and parse the data.
- *Indexes*—A set of indexes created by Lucene.
- *LinkDb*—This directory contains the links associated with each URL, including the source URL and the anchor text of the link.
- *Segments*—There are a number of segments in this directory. Each segment is named based on the date and time that it was created and contains the pages that are fetched by the crawler during a particular run.



**Figure 6.5** The directory structure after the crawl

Next, let's look at statistics associated with the `crawlDb`. Execute the following command:

```
nutch readDb crawl.wiki-ci/crawlDb -stats
```

You should see output similar to that shown in figure 6.6.

```
$ nutch readDb crawl.wiki-ci/crawlDb -stats
CrawlDb statistics start: crawl.wiki-ci/crawlDb
Statistics for CrawlDb: crawl.wiki-ci/crawlDb
TOTAL urls:      1971
retry 0:         1971
min score:       0.0
avg score:       0.0020
max score:       1.005
status 1 (db_unfetched):      1917
status 2 (db_fetched):      50
status 3 (db_gone):          2
status 5 (db_redir_perm):     2
CrawlDb statistics: done
```

**Figure 6.6** The stats associated with the `crawlDb`

It's also helpful to dig deeper into the `crawlDb` and get a dump of the contents in the database. Execute the following command:

```
nutch readDb crawl.wiki-ci/crawlDb -dump crawl.wiki-ci/stats
```

This generates a file `C:\nutch\nutch-0.9\crawl.wiki-ci\stats part-000`, the contents of which will be similar to listing 6.12.

#### Listing 6.12 Dump of the URLs from the `crawlDb`

```
http://ar.wikipedia.org/wiki/%D8%BA%D8%A8%D8%A7%D8%A1 Version: 5
Status: 1 (db_unfetched)
Fetch time: Sat Sep 15 10:32:03 PDT 2007
Modified time: Wed Dec 31 16:00:00 PST 1969
Retries since fetch: 0
Retry interval: 30.0 days
Score: 2.3218017E-4
```



```

Signature: null
Metadata: null

http://ca.wikipedia.org/wiki/M%C3%A8trica_%28matem%C3%A0tiques%29
  Version: 5
Status: 1 (db_unfetched)
Fetch time: Sat Sep 15 10:32:02 PDT 2007
Modified time: Wed Dec 31 16:00:00 PST 1969
Retries since fetch: 0
Retry interval: 30.0 days
Score: 3.1979533E-4
Signature: null
Metadata: null

```

You can look into the contents of the segments in a similar manner:

```

nutch readseg -dump crawl.wiki-ci/
  segments/20070915103026 crawl.wiki-ci/stats/segments

```

This creates a file dump in `C:\nutch\nutch-0.9\crawl.wiki-ci\stats\segments`.

Next, in listing 6.13 let's see how we can search the newly created search index using the Nutch web application.

#### Listing 6.13 Dump of a Nutch segment

```

Recno:: 0
URL:: http://en.wikipedia.org/

CrawlDatum::
Version: 5
Status: 67 (linked)
Fetch time: Sat Sep 15 10:30:33 PDT 2007
Modified time: Wed Dec 31 16:00:00 PST 1969
Retries since fetch: 0
Retry interval: 30.0 days
Score: 0.016949153
Signature: null
Metadata: null

Recno:: 1
URL:: http://en.wikipedia.org/skins-1.5

CrawlDatum::
Version: 5
Status: 67 (linked)
Fetch time: Sat Sep 15 10:30:33 PDT 2007
Modified time: Wed Dec 31 16:00:00 PST 1969
Retries since fetch: 0
Retry interval: 30.0 days
Score: 0.016949153
Signature: null
Metadata: null

```

With this overview, let's next look at how we can set up Nutch to search for the contents that have been crawled.

### 6.3.3 Searching with Nutch

When you unzip your Nutch installation, you should find the `nutch.war` file. Place this war file in your Tomcat webapps directory. The Nutch web application finds its indexes in the `/segments` directory where you start Tomcat, so we need to point Nutch to where we have the crawled data. Therefore, go to

```
C:\apache-tomcat-6.0.13\apache-tomcat-6.0.13\webapps\nutch\WEB-INF\classes\nutch-site.xml
```

Change the contents of this file to those shown in listing 6.14.

#### Listing 6.14 Configuring `nutch-site.xml`

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->
<configuration>
  <property>
    <name>searcher.dir</name>
    <value>C:\nutch\nutch-0.9\crawl.wiki-ci</value>
  </property>
</configuration>
```

Directory where  
Nutch should get data

Now, start up Tomcat (`startup.sh`) and point your browser to the following URL (assuming that Tomcat is running on its default port of 8080):

```
http://localhost:8080/nutch
```

Your browser window should show the Nutch search screen, as shown in figure 6.7.

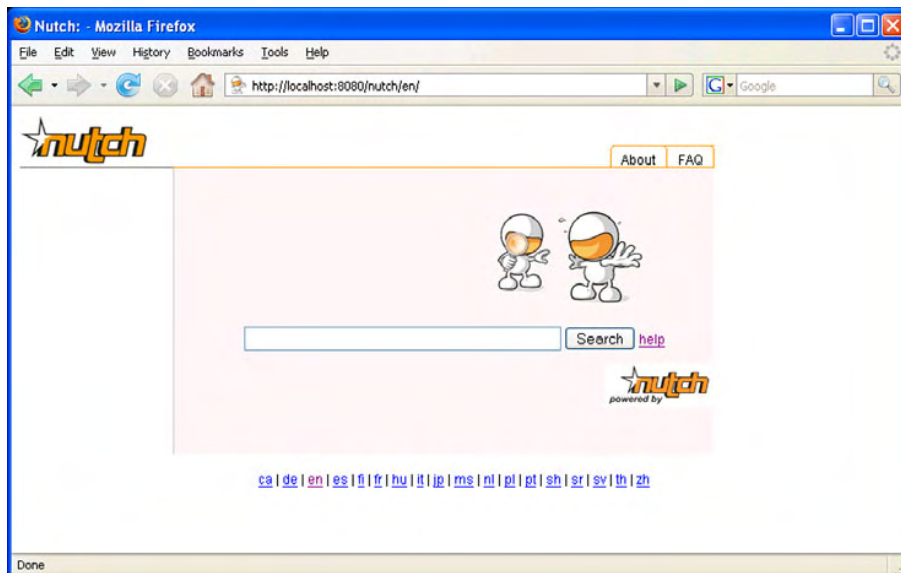
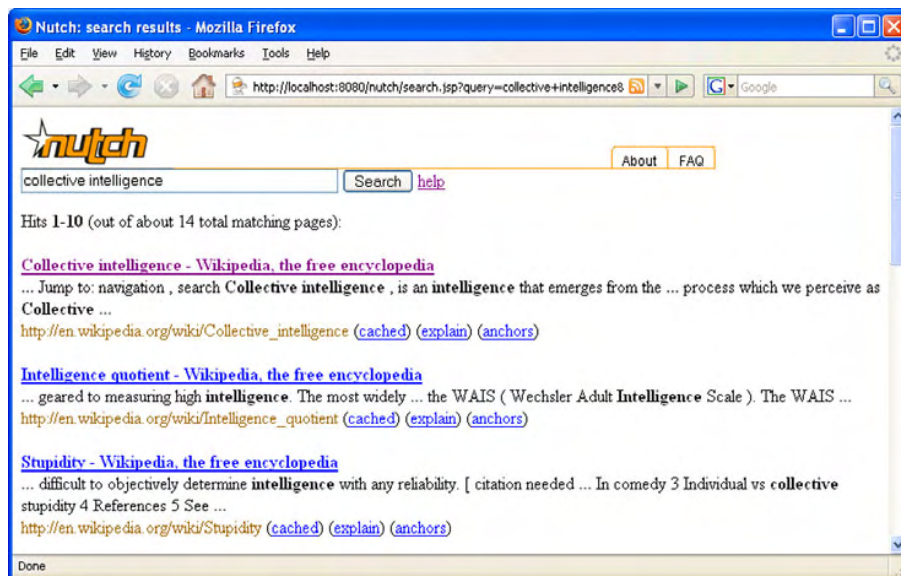


Figure 6.7 The search screen for the Nutch application



**Figure 6.8** Searching for *collective intelligence* using the Nutch search application

Search for the term *collective intelligence* and your browser should look similar to the one in figure 6.8. Play around with the various links, especially the explain and anchors links that are associated with each result.

So far in this section, we've gone through a simple example to crawl Wikipedia, starting with its page on collective intelligence. We've gone through the various directories generated by Nutch and looked at how to use the search tool. This should have given you a good overview of how to use Nutch. I referred you to the various references to set up the system to do a full Internet crawl and maintain the crawled data. Before we end this section, it's useful to briefly go through two important concepts: Apache Hadoop and MapReduce. These are the principles on which Nutch has been built to scale to billions of pages using commodity hardware in a distributed platform.

### 6.3.4 Apache Hadoop, MapReduce, and Dryad

Apache Hadoop is a software platform that lets you write and run applications for processing large datasets using commodity hardware in a distributed platform. Hadoop uses the Hadoop Distributed File System<sup>7</sup> (HDFS) and implements MapReduce.<sup>8</sup>

HDFS is a part of Apache Hadoop project, which in turn is a subproject of Apache Lucene. HDFS is motivated by concepts used in the Google File System.<sup>9</sup> The MapReduce concept has been extensively used by Google and deals with dividing the application into small units of work that can be executed in a distributed environment.

<sup>7</sup> See [http://lucene.apache.org/hadoop/hdfs\\_design.htm](http://lucene.apache.org/hadoop/hdfs_design.htm).

<sup>8</sup> See Dean, J., and Ghemawat, S., MapReduce: Simplified Data Processing on Large Clusters, <http://labs.google.com/papers/mapreduce.html>.

<sup>9</sup> <http://labs.google.com/papers/gfs.html>

Apache Hadoop aims to provide an open source implementation of MapReduce that anyone can use in their own distributed environment.

In the MapReduce paradigm, computation is split into two parts. First, apply a map function to each logical record to generate a set of intermediate key/value pairs. Then apply a reduce operation to compute a final answer that combines all the values for a given key.

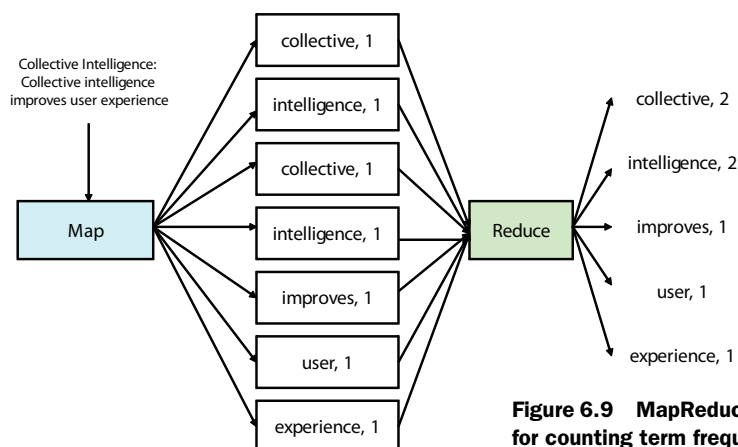
A simple example best illustrates the paradigm, and is illustrated in figure 6.9. Assume that we need to compute the term frequencies associated with the various terms in a page. Using part of the example we looked at in section 4.3, let's assume that a page consists of the following text:

*Collective Intelligence: Collective intelligence improves user experience*

To process this, we would write a map method that is passed, say, an ID for the page as the key and the page text as the value. This method then generates seven intermediate key value pairs, one for each word, as shown in figure 6.9. These intermediate values are processed by the reduce function that counts the values for each of the keys. The output from the reduce function consists of five terms with their associated frequencies. Using this paradigm, a developer doesn't need to worry about distribution; the code is automatically parallelized.

Nutch uses Apache Hadoop and the MapReduce paradigm to scale to crawling and indexing billions of pages. For more details on Hadoop and MapReduce, see the references on this topic.

Microsoft's answer to MapReduce has been the development of Dryad.<sup>10</sup> Dryad is a distributed computing platform developed by Microsoft Research and designed to provide operating system-level abstraction for thousands of PCs in a data center. Like MapReduce, a programmer can leverage parallel processing capabilities of thousands of machines without knowing anything about concurrent programming. With Dryad, you write several sequential programs and then connect them using one-way channels.



**Figure 6.9 MapReduce example for counting term frequencies**

<sup>10</sup> <http://research.microsoft.com/research/sv/dryad/>

The computation can be represented as a directed graph, with each vertex corresponding to a program and channels corresponding to the edges of the graph. A job in Dryad corresponds to traversing a directed acyclic graph, whose structure can change even during execution. Dryad infrastructure manages the creation, management, monitoring, and visualization of jobs. It provides fault-tolerance to machine failures and automatically re-executes failed jobs.

In this section, we looked at using the open source crawler, Nutch, for crawling the web. We also looked at how Nutch can be used for searching through the retrieved content and the various options for building a scalable web crawler.

## 6.4 Summary

Web crawlers are programs that retrieve content from sites by following hyperlinks in the document. Crawlers are useful for retrieving content from external sites. When the crawling process is guided by relevancy, it's called focused or intelligent crawling.

A typical focused crawling process consists of seeding the crawler with some seed URLs. The crawler visits the next available URL, retrieves the content, and measures the relevance of the content to the topic of interest. If the content is acceptable, then it parses the content to extract URLs and in turn visits these URLs.

There are significant costs associated with crawling the entire Web. These include costs for the software and hardware, high-speed network access, storage devices, and administrating the infrastructure. Using a focused crawler can help retrieve relevant information by crawling a subset of available crawling URLs.

With this chapter, we conclude the first part of the book that deals with gathering information from within and outside your application. In the next part, we look at how to analyze this information and build models to make your application more intelligent.

## 6.5 Resources

A Standard for Robots Exclusion. <http://www.robotstxt.org/wc/norobots.html>

Chakrabarti, Soumen. *Mining the Web. Discovering Knowledge from Hypertext Data*. 2005. Morgan Kaufmann Publishers.

Chakrabarti, Soumen, Martin H. Van den Berg, and Byron E. Dom. "Focused Crawling: A New Approach to Topic-Specific Web Resource Discovery." 1999. Proceedings of the 8th International WWW Conference, pp. 545-562.

Cutting, Doug. "MapReduce in Nutch." <http://wiki.apache.org/nutch-data/attachments/Presentations/attachments/mapred.pdf>

Dean, J., and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." <http://labs.google.com/papers/mapreduce.html>

De Bra, Paul, Geert-Jan Houben, Yoram Kornatzky, and Renier Post. "Information Retrieval in Distributed Hypertexts." 1994. Proceedings of the 4th RIAO (Computer Assisted Information Retrieval) Conference, pp. 481-491.

Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google File System." <http://labs.google.com/papers/gfs.html>

Gulli, A., and A. Signorini. "The Indexable Web is more than 11.5 billion pages." 2005. <http://www.cs.uiowa.edu/~asignori/web-size/>

Hadoop. <http://lucene.apache.org/hadoop/>

- “How Google Works.” <http://www.baselinemag.com/article2/0,1540,1985048,00.asp>
- Isard, Michael, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. “Dryad, Distributed Data-Parallel Programs from Sequential Building Blocks.” Eurosys ’07. <http://research.microsoft.com/users/mbudiu/eurosys07.pdf>
- Lucene Hadoop Wiki. <http://labs.google.com/papers/gfs.html>
- “Nutch 0.9 Crawl Script Tutorial.” [http://wiki.apache.org/nutch/Nutch\\_0%2e9\\_Crawl\\_Script\\_Tutorial](http://wiki.apache.org/nutch/Nutch_0%2e9_Crawl_Script_Tutorial)
- Nutch, the Java Search Engine. [http://wiki.apache.org/nutch/Nutch\\_-\\_The\\_Java\\_Search\\_Engine](http://wiki.apache.org/nutch/Nutch_-_The_Java_Search_Engine)
- Nutch Wiki. <http://wiki.apache.org/nutch/>
- NutchHadoop Tutorial. “How to Setup Nutch and Hadoop.” <http://wiki.apache.org/nutch/NutchHadoopTutorial>
- Perez, Juan Carlos, IDG News Service. “Google Counts More Than 1 Trillion Unique Web URLs”, [http://www.pcworld.com/businesscenter/article/148964/google\\_counts\\_more\\_than\\_1\\_trillion\\_unique\\_web\\_urls.html](http://www.pcworld.com/businesscenter/article/148964/google_counts_more_than_1_trillion_unique_web_urls.html)
- “Simple MapReduce Tutorial.” <http://wiki.apache.org/nutch/SimpleMapReduceTutorial>
- Sitemaps. <http://en.wikipedia.org/wiki/Sitemaps>
- “The Hadoop Distributed File System: Architecture and Design.” [http://lucene.apache.org/hadoop/hdfs\\_design.htm](http://lucene.apache.org/hadoop/hdfs_design.htm)
- What are sitemaps? <http://www.sitemaps.org/>
- White, Tom. Tom White’s Blog. “MapReduce.” <http://weblogs.java.net/blog/tomwhite/archive/2005/09/mapreduce.html>
- “Introduction to Nutch, Part 1: Crawling.” <http://today.java.net/pub/a/today/2006/01/10/introduction-to-nutch-1.html>
- “Introduction to Nutch, Part 2: Searching.” <http://today.java.net/pub/a/today/2006/02/16/introduction-to-nutch-2.html>
- Zhuang, Ziming, Rohit Wage, and C. Lee Giles. “What’s There and What’s Not? Focused Crawling for Missing Documents in Digital Libraries.” 2005. JCDL, pp. 301-310.