# 3

# Crawls and Feeds

> "You've stuck your webs into my business for the last time."
>
> Doc Ock, *Spider Man 2*

## 3.1 Deciding What to Search

This book is about the details of building a search engine, from the mathematics behind ranking to the algorithms of query processing. Although we focus heavily on the technology that makes search engines work, and great technology can make a good search engine even better, it is the information in the document collection that makes search engines useful. In other words, if the right documents are not stored in the search engine, no search technique will be able to find relevant information.

The title of this section implies the question, "What should we search?" The simple answer is *everything you possibly can*. Every document answers at least one question (i.e., "Now where was that document again?"), although the best documents answer many more. Every time a search engine adds another document, the number of questions it can answer increases. On the other hand, adding many poor-quality documents increases the burden on the ranking process to find only the best documents to show to the user. Web search engines, however, show how successful search engines can be, even when they contain billions of low-quality documents with little useful content.

Even useful documents can become less useful over time. This is especially true of news and financial information where, for example, many people want to know about today's stock market report, but only a few care about what happened yesterday. The frustration of finding out-of-date web pages and links in a search result list is, unfortunately, a common experience. Search engines are most effective when they contain the most recent information in addition to archives of older material.

This chapter introduces techniques for finding documents to search, whether on the Web, on a file server, on a computer's hard disk, or in an email program. We will discuss strategies for storing documents and keeping those documents up-to-date. Along the way, we will discuss how to pull data out of files, navigating through issues of character encodings, obsolete file formats, duplicate documents, and textual noise. By the end of this chapter you will have a solid grasp on how to get document data into a search engine, ready to be indexed.

## 3.2 Crawling the Web

To build a search engine that searches web pages, you first need a copy of the pages that you want to search. Unlike some of the other sources of text we will consider later, web pages are particularly easy to copy, since they are meant to be retrieved over the Internet by browsers. This instantly solves one of the major problems of getting information to search, which is how to get the data from the place it is stored to the search engine.

Finding and downloading web pages automatically is called *crawling*, and a program that downloads pages is called a *web crawler*.[1] There are some unique challenges to crawling web pages. The biggest problem is the sheer scale of the Web. There are at least tens of billions of pages on the Internet. The "at least" in the last sentence is there because nobody is sure how many pages there are. Even if the number of pages in existence today could be measured exactly, that number would be immediately wrong, because pages are constantly being created. Every time a user adds a new blog post or uploads a photo, another web page is created. Most organizations do not have enough storage space to store even a large fraction of the Web, but web search providers with plenty of resources must still constantly download new content to keep their collections current.

Another problem is that web pages are usually not under the control of the people building the search engine database. Even if you know that you want to copy all the pages from www.company.com, there is no easy way to find out how many pages there are on the site. The owners of that site may not want you to copy some of the data, and will probably be angry if you try to copy it too quickly or too frequently. Some of the data you want to copy may be available only by typing a request into a form, which is a difficult process to automate.

[1] Crawling is also occasionally referred to as *spidering*, and a crawler is sometimes called a *spider*.

### 3.2.1 Retrieving Web Pages

Each web page on the Internet has its own unique *uniform resource locator*, or *URL*. Any URL used to describe a web page has three parts: the scheme, the hostname, and the resource name (Figure 3.1). Web pages are stored on *web servers*, which use a protocol called *Hypertext Transfer Protocol*, or *HTTP*, to exchange information with client software. Therefore, most URLs used on the Web start with the scheme `http`, indicating that the URL represents a resource that can be retrieved using HTTP. The *hostname* follows, which is the name of the computer that is running the web server that holds this web page. In the figure, the computer's name is `www.cs.umass.edu`, which is a computer in the University of Massachusetts Computer Science department. This URL refers to a page on that computer called `/csinfo/people.html`.

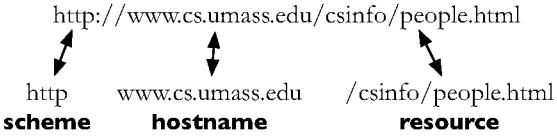


**Fig. 3.1.** A uniform resource locator (URL), split into three parts

Web browsers and web crawlers are two different kinds of web clients, but both fetch web pages in the same way. First, the client program connects to a *domain name system* (DNS) server. The DNS server translates the hostname into an *internet protocol* (IP) address. This IP address is a number that is typically 32 bits long, but some networks now use 128-bit IP addresses. The program then attempts to connect to a server computer with that IP address. Since that server might have many different programs running on it, with each one listening to the network for new connections, each program listens on a different *port*. A port is just a 16-bit number that identifies a particular service. By convention, requests for web pages are sent to port 80 unless specified otherwise in the URL.

Once the connection is established, the client program sends an HTTP request to the web server to request a page. The most common HTTP request type is a GET request, for example:

```
GET /csinfo/people.html HTTP/1.0
```

This simple request asks the server to send the page called `/csinfo/people.html` back to the client, using version 1.0 of the HTTP protocol specification. After

sending a short header, the server sends the contents of that file back to the client. If the client wants more pages, it can send additional requests; otherwise, the client closes the connection.

A client can also fetch web pages using POST requests. A POST request is like a GET request, except that it can send additional request information to the server. By convention, GET requests are used for retrieving data that already exists on the server, whereas POST requests are used to tell the server something. A POST request might be used when you click a button to purchase something or to edit a web page. This convention is useful if you are running a web crawler, since sending only GET requests helps make sure your crawler does not inadvertently order a product.

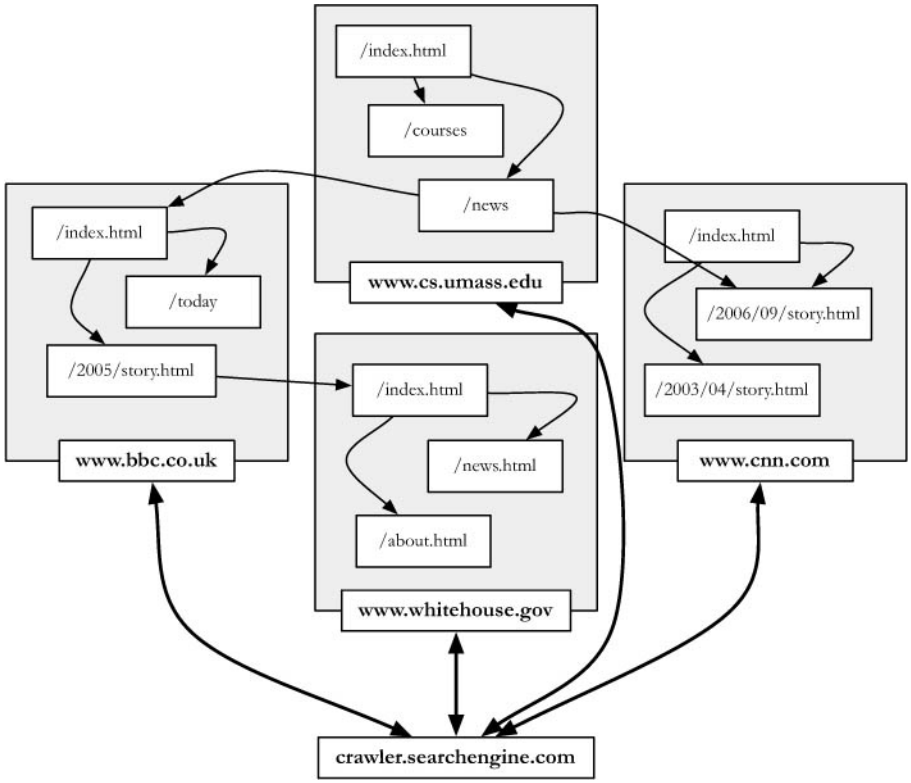


**Fig. 3.2.** Crawling the Web. The web crawler connects to web servers to find pages. Pages may link to other pages on the same server or on different servers.

### 3.2.2 The Web Crawler

Figure 3.2 shows a diagram of the Web from a simple web crawler's perspective. The web crawler has two jobs: downloading pages and finding URLs.

The crawler starts with a set of *seeds*, which are a set of URLs given to it as parameters. These seeds are added to a URL request queue. The crawler starts fetching pages from the request queue. Once a page is downloaded, it is parsed to find link tags that might contain other useful URLs to fetch. If the crawler finds a new URL that it has not seen before, it is added to the crawler's request queue, or *frontier*. The frontier may be a standard queue, or it may be ordered so that important pages move to the front of the list. This process continues until the crawler either runs out of disk space to store pages or runs out of useful links to add to the request queue.

If a crawler used only a single thread, it would not be very efficient. Notice that the web crawler spends a lot of its time waiting for responses: it waits for the DNS server response, then it waits for the connection to the web server to be acknowledged, and then it waits for the web page data to be sent from the server. During this waiting time, the CPU of the web crawler machine is idle and the network connection is unused. To reduce this inefficiency, web crawlers use threads and fetch hundreds of pages at once.

Fetching hundreds of pages at once is good for the person running the web crawler, but not necessarily good for the person running the web server on the other end. Just imagine how the request queue works in practice. When a web page like www.company.com is fetched, it is parsed and all of the links on that page are added to the request queue. The crawler will then attempt to fetch all of those pages at once. If the web server for www.company.com is not very powerful, it might spend all of its time handling requests from the crawler instead of handling requests from real users. This kind of behavior from web crawlers tends to make web server administrators very angry.

To avoid this problem, web crawlers use *politeness policies*. Reasonable web crawlers do not fetch more than one page at a time from a particular web server. In addition, web crawlers wait at least a few seconds, and sometimes minutes, between requests to the same web server. This allows web servers to spend the bulk of their time processing real user requests. To support this, the request queue is logically split into a single queue per web server. At any one time, most of these per-server queues are off-limits for crawling, because the crawler has fetched a page from that server recently. The crawler is free to read page requests only from queues that haven't been accessed within the specified politeness window.

When using a politeness window, the request queue must be very large in order to achieve good performance. Suppose a web crawler can fetch 100 pages each second, and that its politeness policy dictates that it cannot fetch more than one page each 30 seconds from a particular web server. The web crawler needs to have URLs from at least 3,000 different web servers in its request queue in order to achieve high throughput. Since many URLs will come from the same servers, the request queue needs to have tens of thousands of URLs in it before a crawler can reach its peak throughput.

```
User-agent: *
Disallow: /private/
Disallow: /confidential/
Disallow: /other/
Allow: /other/public/

User-agent: FavoredCrawler
Disallow:

Sitemap: http://mysite.com/sitemap.xml.gz
```

**Fig. 3.3.** An example robots.txt file

Even crawling a site slowly will anger some web server administrators who object to any copying of their data. Web server administrators who feel this way can store a file called `/robots.txt` on their web servers. Figure 3.3 contains an example robots.txt file. The file is split into blocks of commands that start with a `User-agent:` specification. The `User-agent:` line identifies a crawler, or group of crawlers, affected by the following rules. Following this line are `Allow` and `Disallow` rules that dictate which resources the crawler is allowed to access. In the figure, the first block indicates that all crawlers need to ignore resources that begin with `/private/`, `/confidential/`, or `/other/`, except for those that begin with `/other/public/`. The second block indicates that a crawler named `FavoredCrawler` gets its own set of rules: it is allowed to copy everything.

The final block of the example is an optional `Sitemap:` directive, which will be discussed later in this section.

Figure 3.4 shows an implementation of a crawling thread, using the crawler building blocks we have seen so far. Assume that the frontier has been initialized

```
procedure CrawlerThread(frontier)
    while not frontier.done() do
        website ← frontier.nextSite()
        url ← website.nextURL()
        if website.permitsCrawl(url) then
            text ← retrieveURL(url)
            storeDocument(url, text)
            for each url in parse(text) do
                frontier.addURL(url)
            end for
        end if
        frontier.releaseSite(website)
    end while
end procedure
```

**Fig. 3.4.** A simple crawling thread implementation

with a few URLs that act as seeds for the crawl. The crawling thread first retrieves a website from the frontier. The crawler then identifies the next URL in the website's queue. In permitsCrawl, the crawler checks to see if the URL is okay to crawl according to the website's robots.txt file. If it can be crawled, the crawler uses retrieveURL to fetch the document contents. This is the most expensive part of the loop, and the crawler thread may block here for many seconds. Once the text has been retrieved, storeDocument stores the document text in a document database (discussed later in this chapter). The document text is then parsed so that other URLs can be found. These URLs are added to the frontier, which adds them to the appropriate website queues. When all this is finished, the website object is returned to the frontier, which takes care to enforce its politeness policy by not giving the website to another crawler thread until an appropriate amount of time has passed. In a real crawler, the timer would start immediately after the document was retrieved, since parsing and storing the document could take a long time.

### 3.2.3 Freshness

Web pages are constantly being added, deleted, and modified. To keep an accurate view of the Web, a web crawler must continually revisit pages it has already crawled to see if they have changed in order to maintain the *freshness* of the document collection. The opposite of a fresh copy is a *stale* copy, which means a copy that no longer reflects the real content of the web page.

Client request:

```
HEAD /csinfo/people.html HTTP/1.1
Host: www.cs.umass.edu
```

Server response:

```
HTTP/1.1 200 OK
Date: Thu, 03 Apr 2008 05:17:54 GMT
Server: Apache/2.0.52 (CentOS)
Last-Modified: Fri, 04 Jan 2008 15:28:39 GMT
ETag: "239c33-2576-2a2837c0"
Accept-Ranges: bytes
Content-Length: 9590
Connection: close
Content-Type: text/html; charset=ISO-8859-1
```

**Fig. 3.5.** An HTTP HEAD request and server response

The HTTP protocol has a special request type called HEAD that makes it easy to check for page changes. The HEAD request returns only header information about the page, but not the page itself. Figure 3.5 contains an example HEAD request and response. The `Last-Modified` value indicates the last time the page content was changed. Notice that the date is also sent along with the response, as well as in response to a GET request. This allows the web crawler to compare the date it received from a previous GET request with the `Last-Modified` value from a HEAD request.

A HEAD request reduces the cost of checking on a page, but does not eliminate it. It simply is not possible to check every page every minute. Not only would that attract more negative reactions from web server administrators, but it would cause enormous load on the web crawler and the incoming network connection.

Thankfully, most web pages are not updated every few minutes. Some of them, like news websites, do change frequently. Others, like a person's home page, change much less often. Even within a page type there can be huge variations in the modification rate. For example, some blogs are updated many times a day, whereas others go months between updates. It does little good to continuously check sites that are rarely updated. Therefore, one of the crawler's jobs is to measure the rate at which each page changes. Over time, this data can be used to estimate how frequently each page changes.

Given that a web crawler can't update every page immediately as it changes, the crawler needs to have some metric for measuring crawl freshness. In this chapter, we've used freshness as a general term, but freshness is also the name of a metric.

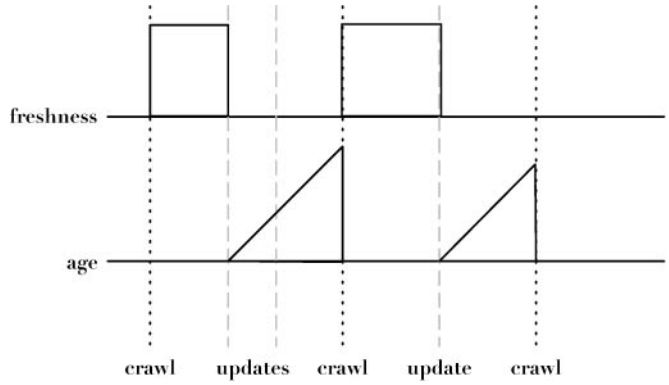


**Fig. 3.6.** Age and freshness of a single page over time

Under the *freshness* metric, a page is *fresh* if the crawl has the most recent copy of a web page, but *stale* otherwise. Freshness is then the fraction of the crawled pages that are currently fresh.

Keeping freshness high seems like exactly what you'd want to do, but optimizing for freshness can have unintended consequences. Suppose that http://www.example.com is a popular website that changes its front page slightly every minute. Unless your crawler continually polls http://www.example.com, you will almost always have a stale copy of that page. Notice that if you want to optimize for freshness, the appropriate strategy is to stop crawling this site completely! If it will never be fresh, it can't help your freshness value. Instead, you should allocate your crawler's resources to pages that change less frequently.

Of course, users will revolt if you decide to optimize your crawler for freshness. They will look at http://www.example.com and wonder why your indexed copy is months out of date.

*Age* is a better metric to use. You can see the difference between age and freshness in Figure 3.6. In the top part of the figure, you can see that pages become fresh immediately when they are crawled, but once the page changes, the crawled page becomes stale. Under the age metric, the page has age 0 until it is changed, and then its age grows until the page is crawled again.

Suppose we have a page with change frequency $\lambda$, meaning that we expect it to change $\lambda$ times in a one-day period. We can calculate the expected age of a page $t$ days after it was last crawled:

$$\text{Age}(\lambda, t) = \int_0^t P(\text{page changed at time } x)(t - x)dx$$

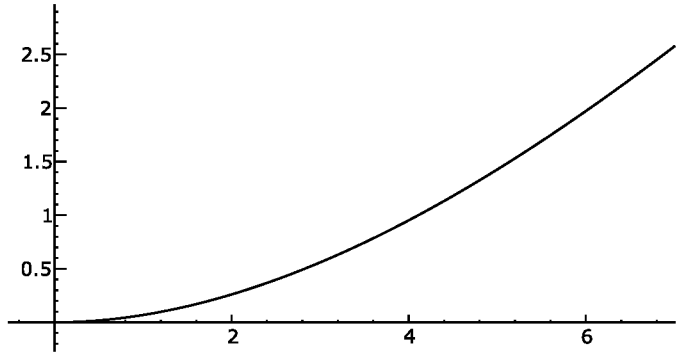


**Fig. 3.7.** Expected age of a page with mean change frequency $\lambda = 1/7$ (one week)

The $(t - x)$ expression is an age: we assume the page is crawled at time $t$, but that it changed at time $x$. We multiply that by the probability that the page actually changed at time $x$. Studies have shown that, on average, web page updates follow the Poisson distribution, meaning that the time until the next update is governed by an exponential distribution (Cho & Garcia-Molina, 2003). This gives us a formula to plug into the $P(\text{page changed at time } x)$ expression:

$$\text{Age}(\lambda, t) = \int_0^t \lambda e^{-\lambda x}(t - x)dx$$

Figure 3.7 shows the result of plotting this expression for a fixed $\lambda = 1/7$, indicating roughly one change a week. Notice how the expected age starts at zero, and rises slowly at first. This is because the page is unlikely to have changed in the first day. As the days go by, the probability that the page has changed increases. By the end of the week, the expected age of the page is about 2.6 days. This means that if your crawler crawls each page once a week, and each page in your collection has a mean update time of once a week, the pages in your index will be 2.6 days old on average just before the crawler runs again.

Notice that the second derivative of the Age function is always positive. That is, the graph is not only increasing, but its rate of increase is always increasing. This positive second derivative means that the older a page gets, the more it costs you to not crawl it. Optimizing this metric will never result in the conclusion that optimizing for freshness does, where sometimes it is economical to not crawl a page at all.

### 3.2.4 Focused Crawling

Some users would like a search engine that focuses on a specific topic of information. For instance, at a website about movies, users might want access to a search engine that leads to more information about movies. If built correctly, this type of *vertical search* can provide higher accuracy than general search because of the lack of extraneous information in the document collection. The computational cost of running a vertical search will also be much less than a full web search, simply because the collection will be much smaller.

The most accurate way to get web pages for this kind of engine would be to crawl a full copy of the Web and then throw out all unrelated pages. This strategy requires a huge amount of disk space and bandwidth, and most of the web pages will be discarded at the end.

A less expensive approach is *focused*, or *topical*, crawling. A focused crawler attempts to download only those pages that are about a particular topic. Focused crawlers rely on the fact that pages about a topic tend to have links to other pages on the same topic. If this were perfectly true, it would be possible to start a crawl at one on-topic page, then crawl all pages on that topic just by following links from a single root page. In practice, a number of popular pages for a specific topic are typically used as seeds.

Focused crawlers require some automatic means for determining whether a page is about a particular topic. Chapter 9 will introduce text classifiers, which are tools that can make this kind of distinction. Once a page is downloaded, the crawler uses the classifier to decide whether the page is on topic. If it is, the page is kept, and links from the page are used to find other related sites. The anchor text in the outgoing links is an important clue of topicality. Also, some pages have more on-topic links than others. As links from a particular web page are visited, the crawler can keep track of the topicality of the downloaded pages and use this to determine whether to download other similar pages. Anchor text data and page link topicality data can be combined together in order to determine which pages should be crawled next.

### 3.2.5 Deep Web

Not all parts of the Web are easy for a crawler to navigate. Sites that are difficult for a crawler to find are collectively referred to as the *deep Web* (also called the *hidden Web*). Some studies have estimated that the deep Web is over a hundred

times larger than the traditionally indexed Web, although it is very difficult to measure this accurately.

Most sites that are a part of the deep Web fall into three broad categories:

- *Private sites* are intentionally private. They may have no incoming links, or may require you to log in with a valid account before using the rest of the site. These sites generally want to block access from crawlers, although some news publishers may still want their content indexed by major search engines.
- *Form results* are sites that can be reached only after entering some data into a form. For example, websites selling airline tickets typically ask for trip information on the site's entry page. You are shown flight information only after submitting this trip information. Even though you might want to use a search engine to find flight timetables, most crawlers will not be able to get through this form to get to the timetable information.
- *Scripted pages* are pages that use JavaScript™, Flash®, or another client-side language in the web page. If a link is not in the raw HTML source of the web page, but is instead generated by JavaScript code running on the browser, the crawler will need to execute the JavaScript on the page in order to find the link. Although this is technically possible, executing JavaScript can slow down the crawler significantly and adds complexity to the system.

Sometimes people make a distinction between *static pages* and *dynamic pages*. Static pages are files stored on a web server and displayed in a web browser unmodified, whereas dynamic pages may be the result of code executing on the web server or the client. Typically it is assumed that static pages are easy to crawl, while dynamic pages are hard. This is not quite true, however. Many websites have dynamically generated web pages that are easy to crawl; wikis are a good example of this. Other websites have static pages that are impossible to crawl because they can be accessed only through web forms.

Web administrators of sites with form results and scripted pages often want their sites to be indexed, unlike the owners of private sites. Of these two categories, scripted pages are easiest to deal with. The site owner can usually modify the pages slightly so that links are generated by code on the server instead of by code in the browser. The crawler can also run page JavaScript, or perhaps Flash as well, although these can take a lot of time.

The most difficult problems come with form results. Usually these sites are repositories of changing data, and the form submits a query to a database system. In the case where the database contains millions of records, the site would need to

expose millions of links to a search engine's crawler. Adding a million links to the front page of such a site is clearly infeasible. Another option is to let the crawler guess what to enter into forms, but it is difficult to choose good form input. Even with good guesses, this approach is unlikely to expose all of the hidden data.

### 3.2.6 Sitemaps

As you can see from the last two sections, the biggest problems in crawling arise because site owners cannot adequately tell crawlers about their sites. In section 3.2.3, we saw how crawlers have to make guesses about when pages will be updated because polling is costly. In section 3.2.5, we saw that site owners sometimes have data that they would like to expose to a search engine, but they can't because there is no reasonable place to store the links. *Sitemaps* solve both of these problems.

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.company.com/</loc>
    <lastmod>2008-01-15</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.7</priority>
  </url>
  <url>
    <loc>http://www.company.com/items?item=truck</loc>
    <changefreq>weekly</changefreq>
  </url>
  <url>
    <loc>http://www.company.com/items?item=bicycle</loc>
    <changefreq>daily</changefreq>
  </url>
</urlset>
```

Fig. 3.8. An example sitemap file

A robots.txt file can contain a reference to a sitemap, like the one shown in Figure 3.8. A sitemap contains a list of URLs and data about those URLs, such as modification time and modification frequency.

There are three URL entries shown in the example sitemap. Each one contains a URL in a `loc` tag. The `changefreq` tag indicates how often this resource is likely to change. The first entry includes a `lastmod` tag, which indicates the last time it was changed. The first entry also includes a `priority` tag with a value of 0.7, which is higher than the default of 0.5. This tells crawlers that this page is more important than other pages on this site.

Why would a web server administrator go to the trouble to create a sitemap? One reason is that it tells search engines about pages it might not otherwise find. Look at the second and third URLs in the sitemap. Suppose these are two product pages. There may not be any links on the website to these pages; instead, the user may have to use a search form to get to them. A simple web crawler will not attempt to enter anything into a form (although some advanced crawlers do), and so these pages would be invisible to search engines. A sitemap allows crawlers to find this hidden content.

The sitemap also exposes modification times. In the discussion of page freshness, we mentioned that a crawler usually has to guess when pages are likely to change. The `changefreq` tag gives the crawler a hint about when to check a page again for changes, and the `lastmod` tag tells the crawler when a page has changed. This helps reduce the number of requests that the crawler sends to a website without sacrificing page freshness.

### 3.2.7 Distributed Crawling

For crawling individual websites, a single computer is sufficient. However, crawling the entire Web requires many computers devoted to crawling. Why would a single crawling computer not be enough? We will consider three reasons.

One reason to use multiple computers is to put the crawler *closer to the sites it crawls*. Long-distance network connections tend to have lower throughput (fewer bytes copied per second) and higher latency (bytes take longer to cross the network). Decreased throughput and increased latency work together to make each page request take longer. As throughput drops and latency rises, the crawler has to open more connections to copy pages at the same rate.

For example, suppose a crawler has a network connection that can transfer 1MB each second. With an average web page size of 20K, it can copy 50 pages each second. If the sites that are being crawled are close, the data transfer rate from them may be 1MB a second. However, it can take 80ms for the site to start sending data, because there is some transmission delay in opening the connection

and sending the request. Let's assume each request takes 100ms (80ms of latency and 20ms of data transfer). Multiplying 50 by 100ms, we see that there is 5 seconds of waiting involved in transferring 50 pages. This means that five connections will be needed to transfer 50 pages in one second. If the sites are farther away, with an average throughput of 100K per second and 500ms of latency, then each request would now take 600ms. Since 50 × 600ms = 30 seconds, the crawler would need to keep 30 connections open to transfer pages at the same rate.

Another reason for multiple crawling computers is to reduce the *number of sites the crawler has to remember*. A crawler has to remember all of the URLs it has already crawled, and all of the URLs that it has queued to crawl. These URLs must be easy to access, because every page that is crawled contains new links that need to be added to the crawl queue. Since the crawler's queue should not contain duplicates or sites that have already been crawled, each new URL must be checked against everything in the queue and everything that has been crawled. The data structure for this lookup needs to be in RAM; otherwise, the computer's crawl speed will be severely limited. Spreading crawling duties among many computers reduces this bookkeeping load.

Yet another reason is that crawling can use a lot of *computing resources*, including CPU resources for parsing and network bandwidth for crawling pages. Crawling a large portion of the Web is too much work for a single computer to handle.

A distributed crawler is much like a crawler on a single computer, except instead of a single queue of URLs, there are many queues. The distributed crawler uses a hash function to assign URLs to crawling computers. When a crawler sees a new URL, it computes a hash function on that URL to decide which crawling computer is responsible for it. These URLs are gathered in batches, then sent periodically to reduce the network overhead of sending a single URL at a time.

The hash function should be computed on just the host part of each URL. This assigns all the URLs for a particular host to a single crawler. Although this may promote imbalance since some hosts have more pages than others, politeness rules require a time delay between URL fetches to the same host. It is easier to maintain that kind of delay by using the same crawling computers for all URLs for the same host. In addition, we would expect that sites from domain.com will have lots of links to other pages on domain.com. By assigning domain.com to a single crawl host, we minimize the number of URLs that need to be exchanged between crawling computers.

## 3.3 Crawling Documents and Email

Even though the Web is a tremendous information resource, a huge amount of digital information is not stored on websites. In this section, we will consider information that you might find on a normal desktop computer, such as email, word processing documents, presentations, or spreadsheets. This information can be searched using a *desktop search* tool. In companies and organizations, *enterprise search* will make use of documents on file servers, or even on employee desktop computers, in addition to local web pages.

Many of the problems of web crawling change when we look at desktop data. In web crawling, just finding the data can be a struggle. On a desktop computer, the interesting data is stored in a file system with familiar semantics. Finding all the files on a hard disk is not particularly difficult, since file systems have directories that are easy to discover. In some ways, a file system is like a web server, but with an automatically generated sitemap.

There are unique challenges in crawling desktop data, however. The first concerns update speed. In desktop search applications, users demand search results based on the current content of their files. This means, for example, being able to search for an email the instant it is received, and being able to search for a document as soon as it has been saved. Notice that this is a much different expectation than with web search, where users can tolerate crawling delays of hours or days. Crawling the file system every second is impractical, but modern file systems can send change notifications directly to the crawler process so that it can copy new files immediately. Remote file systems from file servers usually do not provide this kind of change notification, and so they must be crawled just like a web server.

Disk space is another concern. With a web crawler, we assume that we need to keep a copy of every document that is found. This is less true on a desktop system, where the documents are already stored locally, and where users will be unhappy if a large proportion of the hard disk is taken by the indexer. A desktop crawler instead may need to read documents into memory and send them directly to the indexer. We will discuss indexing more in Chapter 5.

Since websites are meant to be viewed with web browsers, most web content is stored in HTML. On the other hand, each desktop program—the word processor, presentation tool, email program, etc.—has its own file format. So, just finding these files is not enough; eventually they will need to be converted into a format that the indexer can understand. In section 3.5 we will revisit this conversion issue.

Finally, and perhaps most importantly, crawling desktop data requires a focus on data privacy. Desktop systems can have multiple users with different accounts, and user $A$ should not be able to find emails from user $B$'s account through the search feature. This is especially important when we consider crawling shared network file systems, as in a corporate network. The file access permissions of each file must be recorded along with the crawled data, and must be kept up-to-date.

## 3.4 Document Feeds

In general Web or desktop crawling, we assume that any document can be created or modified at any time. However, many documents are *published*, meaning that they are created at a fixed time and rarely updated again. News articles, blog posts, press releases, and email are some of the documents that fit this publishing model. Most information that is time-sensitive is published.

Since each published document has an associated time, published documents from a single source can be ordered in a sequence called a *document feed*. A document feed is particularly interesting for crawlers, since the crawler can easily find all the new documents by examining only the end of the feed.

We can distinguish two kinds of document feeds, *push* and *pull*. A *push* feed alerts the subscriber to new documents. This is like a telephone, which alerts you to an incoming phone call; you don't need to continually check the phone to see if someone is calling. A *pull* feed requires the subscriber to check periodically for new documents; this is like checking your mailbox for new mail to arrive. News feeds from commercial news agencies are often push feeds, but pull feeds are overwhelmingly popular for free services. We will focus primarily on pull feeds in this section.

The most common format for pull feeds is called *RSS*. RSS has at least three definitions: Really Simple Syndication, RDF Site Summary, or Rich Site Summary. Not surprisingly, RSS also has a number of slightly incompatible implementations, and a similar competing format exists called the *Atom Syndication Format*. The proliferation of standards is the result of an idea that gained popularity too quickly for developers to agree on a single standard.

Figure 3.9 shows an RSS 2.0 feed from an example site called http://www.search-engine-news.org. This feed contains two articles: one is about an upcoming SIGIR conference, and the other is about a textbook. Notice that each entry contains a time indicating when it was published. In addition, near the top of the RSS feed there is an tag named `ttl`, which means *time to live*, measured in minutes. This

```
<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Search Engine News</title>
    <link>http://www.search-engine-news.org/</link>
   <description>News about search engines.</description>
    <language>en-us</language>
    <pubDate>Tue, 19 Jun 2008 05:17:00 GMT</pubDate>
    <ttl>60</ttl>

    <item>
      <title>Upcoming SIGIR Conference</title>
      <link>http://www.sigir.org/conference</link>
     <description>The annual SIGIR conference is coming!
        Mark your calendars and check for cheap
        flights.</description>
      <pubDate>Tue, 05 Jun 2008 09:50:11 GMT</pubDate>
      <guid>http://search-engine-news.org#500</guid>
    </item>

    <item>
      <title>New Search Engine Textbook</title>
      <link>http://www.cs.umass.edu/search-book</link>
      <description>A new textbook about search engines
        will be published soon.</description>
      <pubDate>Tue, 05 Jun 2008 09:33:01 GMT</pubDate>
      <guid>http://search-engine-news.org#499</guid>
    </item>
  </channel>
</rss>
```

Fig. 3.9. An example RSS 2.0 feed

feed states that its contents should be cached only for 60 minutes, and information more than an hour old should be considered stale. This gives a crawler an indication of how often this feed file should be crawled.

RSS feeds are accessed just like a traditional web page, using HTTP GET requests to web servers that host them. Therefore, some of the crawling techniques we discussed before apply here as well, such as using HTTP HEAD requests to detect when RSS feeds change.

From a crawling perspective, document feeds have a number of advantages over traditional pages. Feeds give a natural structure to data; even more than with a sitemap, a web feed implies some relationship between the data items. Feeds are easy to parse and contain detailed time information, like a sitemap, but also include a description field about each page (and this description field sometimes contains the entire text of the page referenced in the URL). Most importantly, like a sitemap, feeds provide a single location to look for new data, instead of having to crawl an entire site to find a few new documents.

## 3.5 The Conversion Problem

Search engines are built to search through text. Unfortunately, text is stored on computers in hundreds of incompatible file formats. Standard text file formats include raw text, RTF, HTML, XML, Microsoft Word, ODF (Open Document Format) and PDF (Portable Document Format). There are tens of other less common word processors with their own file formats. But text documents aren't the only kind of document that needs to be searched; other kinds of files also contain important text, such as PowerPoint slides and Excel® spreadsheets. In addition to all of these formats, people often want to search old documents, which means that search engines may need to support obsolete file formats. It is not uncommon for a commercial search engine to support more than a hundred file types.

The most common way to handle a new file format is to use a conversion tool that converts the document content into a tagged text format such as HTML or XML. These formats are easy to parse, and they retain some of the important formatting information (font size, for example). You can see this on any major web search engine. Search for a PDF document, but then click on the "Cached" link at the bottom of a search result. You will be taken to the search engine's view of the page, which is usually an HTML rendition of the original document. For some document types, such as PowerPoint, this cached version can be nearly unreadable. Fortunately, readability isn't the primary concern of the search engine.

The point is to copy this data into the search engine so that it can be indexed and retrieved. However, translating the data into HTML has an advantage: the user does not need to have an application that can read the document's file format in order to view it. This is critical for obsolete file formats.

Documents could be converted to plain text instead of HTML or XML. However, doing this would strip the file of important information about headings and font sizes that could be useful to the indexer. As we will see later, headings and bold text tend to contain words that describe the document content well, so we want to give these words preferential treatment during scoring. Accurate conversion of formatting information allows the indexer to extract these important features.

### 3.5.1 Character Encodings

Even HTML files are not necessarily compatible with each other because of *character encoding* issues. The text that you see on this page is a series of little pictures we call *letters* or *glyphs*. Of course, a computer file is a stream of bits, not a collection of pictures. A character encoding is a mapping between bits and glyphs. For English, the basic character encoding that has been around since 1963 is ASCII. ASCII encodes 128 letters, numbers, special characters, and control characters in 7 bits, extended with an extra bit for storage in bytes. This scheme is fine for the English alphabet of 26 letters, but there are many other languages, and some of those have many more glyphs. The Chinese language, for example, has more than 40,000 characters, with over 3,000 in common use. For the CJK (Chinese-Japanese-Korean) family of East Asian languages, this led to the development of a number of different 2-byte standards. Other languages, such as Hindi or Arabic, also have a range of different encodings. Note that not all encodings even agree on English. The EBCDIC encoding used on mainframes, for example, is completely different than the ASCII encoding used by personal computers.

The computer industry has moved slowly in handling complicated character sets such as Chinese and Arabic. Until recently, the typical approach was to use different language-specific encodings, sometimes called *code pages*. The first 128 values of each encoding are reserved for typical English characters, punctuation, and numbers. Numbers above 128 are mapped to glyphs in the target language, from Hebrew to Arabic. However, if you use a different encoding for each language, you can't write in Hebrew and Japanese in the same document. Additionally, the text itself is no longer self-describing. It's not enough to just store data in a text file; you must also record what encoding was used.

To solve this mess of encoding issues, *Unicode* was developed. Unicode is a single mapping from numbers to glyphs that attempts to include all glyphs in common use in all known languages. This solves the problem of using multiple languages in a single file. Unfortunately, it does not fully solve the problems of binary encodings, because Unicode is a mapping between numbers and glyphs, not bits and glyphs. It turns out that there are many ways to translate Unicode numbers to glyphs! Some of the most popular include UTF-8, UTF-16, UTF-32, and UCS-2 (which is deprecated).

The proliferation of encodings comes from a need for compatibility and to save space. Encoding English text in UTF-8 is identical to the ASCII encoding. Each ASCII letter requires just one byte. However, some traditional Chinese characters can require as many as 4 bytes. The trade-off for compactness for Western languages is that each character requires a variable number of bytes, which makes it difficult to quickly compute the number of characters in a string or to jump to a random location in a string. By contrast, UTF-32 (also known as UCS-4) uses exactly 4 bytes for every character. Jumping to the twentieth character in a UTF-32 string is easy: just jump to the eightieth byte and start reading. Unfortunately, UTF-32 strings are incompatible with all old ASCII software, and UTF-32 files require four times as much space as UTF-8. Because of this, many applications use UTF-32 as their internal text encoding (where random access is important), but use UTF-8 to store text on disk.

| Decimal | Hexadecimal | Encoding |
|---|---|---|
| 0–127 | 0–7F | `0xxxxxxx` |
| 128–2047 | 80–7FF | `110xxxxx 10xxxxxx` |
| 2048–55295 | 800–D7FF | `1110xxxx 10xxxxxx 10xxxxxx` |
| 55296–57343 | D800–DFFF | Undefined |
| 57344–65535 | E000–FFFF | `1110xxxx 10xxxxxx 10xxxxxx` |
| 65536–1114111 | 10000–10FFFF | `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx` |

**Table 3.1.** UTF-8 encoding

Table 3.1 shows an encoding table for UTF-8. The left columns represent ranges of decimal values, and the rightmost column shows how these values are encoded in binary. The `x` characters represent binary digits. For example, the Greek letter pi ($\pi$) is Unicode symbol number 960. In binary, that number is `00000111 11000000` (3C0 in hexadecimal). The second row of the table tells us

that this letter will require 2 bytes to encode in UTF-8. The high 5 bits of the character go in the first byte, and the next 6 bits go in the second byte. The final encoding is **110**11111 **10**000000 (CF80 in hexadecimal). The bold binary digits are the same as the digits from the table, while the x letters from the table have been filled in by binary digits from the Unicode number.

## 3.6 Storing the Documents

After documents have been converted to some common format, they need to be stored in preparation for indexing. The simplest document storage is no document storage, and for some applications this is preferable. In desktop search, for example, the documents are already stored in the file system and do not need to be copied elsewhere. As the crawling process runs, it can send converted documents immediately to an indexing process. By not storing the intermediate converted documents, desktop search systems can save disk space and improve indexing latency.

Most other kinds of search engines need to store documents somewhere. Fast access to the document text is required in order to build document snippets[2] for each search result. These snippets of text give the user an idea of what is inside the retrieved document without actually needing to click on a link.

Even if snippets are not necessary, there are other reasons to keep a copy of each document. Crawling for documents can be expensive in terms of both CPU and network load. It makes sense to keep copies of the documents around instead of trying to fetch them again the next time you want to build an index. Keeping old documents allows you to use HEAD requests in your crawler to save on bandwidth, or to crawl only a subset of the pages in your index.

Finally, document storage systems can be a starting point for information extraction (described in Chapter 4). The most pervasive kind of information extraction happens in web search engines, which extract anchor text from links to store with target web documents. Other kinds of extraction are possible, such as identifying names of people or places in documents. Notice that if information extraction is used in the search application, the document storage system should support modification of the document data.

We now discuss some of the basic requirements for a document storage system, including random access, compression, and updating, and consider the relative

[2] We discuss snippet generation in Chapter 6.

benefits of using a database system or a customized storage system such as Google's BigTable.

### 3.6.1 Using a Database System

If you have used a relational database before, you might be thinking that a database would be a good place to store document data. For many applications, in fact, a database is an excellent place to store documents. A database takes care of the difficult details of storing small pieces of data, such as web pages, and makes it easy to update them later. Most databases also run as a network server, so that the documents are easily available on the network. This could support, for example, a single computer serving documents for snippets while many other computers handle queries. Databases also tend to come with useful import and analysis tools that can make it easier to manage the document collection.

Many companies that run web search engines are reluctant to talk about their internal technologies. However, it appears that few, if any, of the major search engines use conventional relational databases to store documents. One problem is the sheer volume of document data, which can overwhelm traditional database systems. Database vendors also tend to expect that database servers will use the most expensive disk systems, which is impractical given the collection size. We discuss an alternative to a relational database at the end of this section that addresses some of these concerns.

### 3.6.2 Random Access

To retrieve documents quickly in order to compute a snippet for a search result, the document store needs to support random access. Compared to a full relational database, however, only a relatively simple lookup criterion is needed. We want a data store such that we can request the content of a document based on its URL.

The easiest way to handle this kind of lookup is with hashing. Using a hash function on the URL gives us a number we can use to find the data. For small installations, the hash function can tell us which file contains the document. For larger installations, the hash function tells us which server contains the document. Once the document location has been narrowed down to a single file, a B-Tree or sorted data structure can be used to find the offset of the document data within the file.

### 3.6.3 Compression and Large Files

Regardless of whether the application requires random access to documents, the document storage system should make use of large files and compression.

Even a document that seems long to a person is small by modern computer standards. For example, this chapter is approximately 10,000 words, and those words require about 70K of disk space to store. That is far bigger than the average web page, but a modern hard disk can transfer 70K of data in about a millisecond. However, the hard disk might require 10 milliseconds to seek to that file in order to start reading. This is why storing each document in its own file is not a very good idea; reading these small files requires a substantial overhead to open them. A better solution is to store many documents in a single file, and for that file to be large enough that transferring the file contents takes much more time than seeking to the beginning. A good size choice might be in the hundreds of megabytes. By storing documents close together, the indexer can spend most of its time reading data instead of seeking for it.

The Galago search engine includes parsers for three compound document formats: ARC, TREC Text, and TREC Web. In each format, many text documents are stored in the same file, with short regions of document metadata separating the documents. Figure 3.10 shows an example of the TREC Web format. Notice that each document block begins with a `<DOC>` tag and ends with a `</DOC>` tag. At the beginning of the document, the `<DOCHDR>` tag marks a section containing the information about the page request, such as its URL, the date it was crawled, and the HTTP headers returned by the web server. Each document record also contains a `<DOCNO>` field that includes a unique identifier for the document.

Even though large files make sense for data transfer from disk, reducing the total storage requirements for document collections has obvious advantages. Fortunately, text written by people is highly redundant. For instance, the letter $q$ is almost always followed by the letter $u$. Shannon (1951) showed that native English speakers are able to guess the next letter of a passage of English text with 69% accuracy. HTML and XML tags are even more redundant. *Compression* techniques exploit this redundancy to make files smaller without losing any of the content. We will cover compression as it is used for document indexing in Chapter 5, in part because compression for indexing is rather specialized. While research continues into text compression, popular algorithms like DEFLATE (Deutsch, 1996) and LZW (Welch, 1984) can compress HTML and XML text by 80%. This space savings reduces the cost of storing a lot of documents, and also reduces

```
<DOC>
<DOCNO>WTX001-B01-10</DOCNO>
<DOCHDR>
http://www.example.com/test.html 204.244.59.33 19970101013145 text/html 440
HTTP/1.0 200 OK
Date: Wed, 01 Jan 1997 01:21:13 GMT
Server: Apache/1.0.3
Content-type: text/html
Content-length: 270
Last-modified: Mon, 25 Nov 1996 05:31:24 GMT
</DOCHDR>
<HTML>
<TITLE>Tropical Fish Store</TITLE>
Coming soon!
</HTML>
</DOC>
<DOC>
<DOCNO>WTX001-B01-109</DOCNO>
<DOCHDR>
http://www.example.com/fish.html 204.244.59.33 19970101013149 text/html 440
HTTP/1.0 200 OK
Date: Wed, 01 Jan 1997 01:21:19 GMT
Server: Apache/1.0.3
Content-type: text/html
Content-length: 270
Last-modified: Mon, 25 Nov 1996 05:31:24 GMT
</DOCHDR>
<HTML>
<TITLE>Fish Information</TITLE>
This page will soon contain interesting
information about tropical fish.
</HTML>
</DOC>
```

**Fig. 3.10.** An example of text in the TREC Web compound document format

the amount of time it takes to read a document from the disk since there are fewer bytes to read.

Compression works best with large blocks of data, which makes it a good fit for big files with many documents in them. However, it is not necessarily a good idea to compress the entire file as a single block. Most compression methods do not allow random access, so each block can only be decompressed sequentially. If you want random access to the data, it is better to consider compressing in smaller blocks, perhaps one block per document, or one block for a few documents. Small blocks reduce compression ratios (the amount of space saved) but improve request latency.

### 3.6.4 Update

As new versions of documents come in from the crawler, it makes sense to update the document store. The alternative is to create an entirely new document store by merging the new, changed documents from the crawler with document data from the old document store for documents that did not change. If the document data does not change very much, this merging process will be much more expensive than updating the data in place.

```
<a href="http://example.com" >Example website</a>
```

**Fig. 3.11.** An example link with anchor text

Another important reason to support update is to handle anchor text. Figure 3.11 shows an example of anchor text in an HTML link tag. The HTML code in the figure will render in the web browser as a link, with the text Example website that, when clicked, will direct the user to http://example.com. Anchor text is an important feature because it provides a concise summary of what the target page is about. If the link comes from a different website, we may also believe that the summary is unbiased, which also helps us rank documents (see Chapters 4 and 7).

Collecting anchor text properly is difficult because the anchor text needs to be associated with the target page. A simple way to approach this is to use a data store that supports update. When a document is found that contains anchor text, we find the record for the target page and update the anchor text portion of the record. When it is time to index the document, the anchor text is all together and ready for indexing.

### 3.6.5 BigTable

Although a database can perform the duties of a document data store, the very largest document collections demand custom document storage systems. BigTable is the most well known of these systems (Chang et al., 2006). BigTable is a working system in use internally at Google, although at least two open source projects are taking a similar approach. In the next few paragraphs, we will look at the BigTable architecture to see how the problem of document storage influenced its design.

BigTable is a distributed database system originally built for the task of storing web pages. A BigTable instance really is a *big* table; it can be over a petabyte in size, but each database contains only one table. The table is split into small pieces, called *tablets*, which are served by thousands of machines (Figure 3.12).

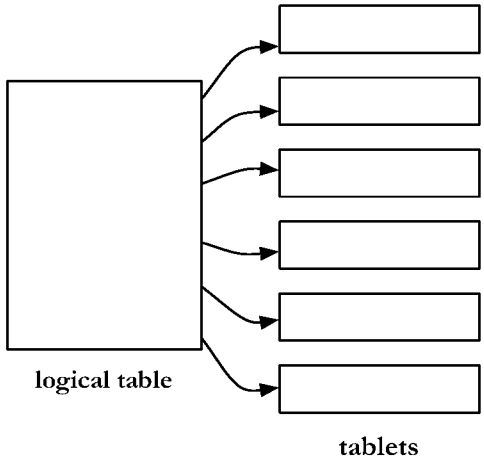


**Fig. 3.12.** BigTable stores data in a single logical table, which is split into many smaller tablets

If you are familiar with relational databases, you will have encountered SQL (Structured Query Language). SQL allows users to write complex and computationally expensive queries, and one of the tasks of the database system is to optimize the processing of these queries to make them as fast as possible. Because some of these queries could take a very long time to complete, a large relational database requires a complex locking system to ensure that the many users of the database do not corrupt it by reading or writing data simultaneously. Isolating users from each other is a difficult job, and many papers and books have been written about how to do it well.

The BigTable approach is quite different. There is no query language, and therefore no complex queries, and it includes only row-level transactions, which would be considered rather simple by relational database standards. However, the simplicity of the model allows BigTable to scale up to very large database sizes while using inexpensive computers, even though they may be prone to failure.

Most of the engineering in BigTable involves failure recovery. The tablets, which are the small sections of the table, are stored in a replicated file system that is accessible by all BigTable tablet servers. Any changes to a BigTable tablet are recorded to a transaction log, which is also stored in a shared file system. If any tablet server crashes, another server can immediately read the tablet data and transaction log from the file system and take over.

Most relational databases store their data in files that are constantly modified. In contrast, BigTable stores its data in immutable (unchangeable) files. Once file data is written to a BigTable file, it is never changed. This also helps in failure recovery. In relational database systems, failure recovery requires a complex series of operations to make sure that files were not corrupted because only some of the outstanding writes completed before the computer crashed. In BigTable, a file is either incomplete (in which case it can be thrown away and re-created from other BigTable files and the transaction log), or it is complete and therefore is not corrupt. To allow for table updates, the newest data is stored in RAM, whereas older data is stored in a series of files. Periodically the files are merged together to reduce the total number of disk files.

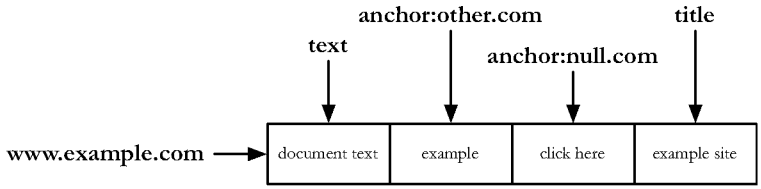


**Fig. 3.13.** A BigTable row

BigTables are logically organized by rows (Figure 3.13). In the figure, the row stores the data for a single web page. The URL, www.example.com, is the row key, which can be used to find this row. The row has many columns, each with a unique name. Each column can have many different timestamps, although that is not shown in the figure. The combination of a row key, a column key, and a times-

tamp point to a single *cell* in the row. The cell holds a series of bytes, which might be a number, a string, or some other kind of data.

In the figure, notice that there is a text column for the full text of the document as well as a title column, which makes it easy to quickly find the document title without parsing the full document text. There are two columns for anchor text. One, called anchor:other.com, includes anchor text from a link from the site other.com to example.com; the text of the link is "example", as shown in the cell. The anchor:null.com describes a link from null.com to example.com with anchor text "click here". Both of these columns are in the anchor *column group*. Other columns could be added to this column group to add information about more links.

BigTable can have a huge number of columns per row, and while all rows have the same column groups, not all rows have the same columns. This is a major departure from traditional database systems, but this flexibility is important, in part because of the lack of tables. In a relational database system, the anchor columns would be stored in one table and the document text in another. Because BigTable has just one table, all the anchor information needs to be packed into a single record. With all the anchor data stored together, only a single disk read is necessary to read all of the document data. In a two-table relational database, at least two reads would be necessary to retrieve this data.

Rows are partitioned into tablets based on their row keys. For instance, all URLs beginning with a could be located in one tablet, while all those starting with b could be in another tablet. Using this kind of range-based partitioning makes it easy for a client of BigTable to determine which server is serving each row. To look up a particular row, the client consults a list of row key ranges to determine which tablet would hold the desired row. The client then contacts the appropriate tablet server to fetch the row. The row key ranges are cached in the client, so that most of the network traffic is between clients and tablet servers.

BigTable's architecture is designed for speed and scale through massive numbers of servers, and for economy by using inexpensive computers that are expected to fail. In order to achieve these goals, BigTable sacrifices some key relational database features, such as a complex query language and multiple-table databases. However, this architecture is well suited for the task of storing and finding web pages, where the primary task is efficient lookups and updates on individual rows.

## 3.7 Detecting Duplicates

Duplicate and *near-duplicate* documents occur in many situations. Making copies and creating new versions of documents is a constant activity in offices, and keeping track of these is an important part of information management. On the Web, however, the situation is more extreme. In addition to the normal sources of duplication, *plagiarism* and *spam* are common, and the use of multiple URLs to point to the same web page and *mirror sites* can cause a crawler to generate large numbers of duplicate pages. Studies have shown that about 30% of the web pages in a large crawl are exact or near duplicates of pages in the other 70% (e.g., Fetterly et al., 2003).

Documents with very similar content generally provide little or no new information to the user, but consume significant resources during crawling, indexing, and search. In response to this problem, algorithms for detecting duplicate documents have been developed so that they can be removed or treated as a group during indexing and ranking.

Detecting exact duplicates is a relatively simple task that can be done using *checksumming* techniques. A checksum is a value that is computed based on the content of the document. The most straightforward checksum is a sum of the bytes in the document file. For example, the checksum for a file containing the text "Tropical fish" would be computed as follows (in hex):

| T | r | o | p | i | c | a | l | | f | i | s | h | *Sum* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 54 | 72 | 6F | 70 | 69 | 63 | 61 | 6C | 20 | 66 | 69 | 73 | 68 | 508 |

Any document file containing the same text would have the same checksum. Of course, any document file containing text that happened to have the same checksum would also be treated as a duplicate. A file containing the same characters in a different order would have the same checksum, for example. More sophisticated functions, such as a *cyclic redundancy check* (CRC), have been developed that consider the positions of the bytes.

The detection of near-duplicate documents is more difficult. Even defining a near-duplicate is challenging. Web pages, for example, could have the same text content but differ in the advertisements, dates, or formatting. Other pages could have small differences in their content from revisions or updates. In general, a near-duplicate is defined using a threshold value for some similarity measure between pairs of documents. For example, a document $D_1$ could be defined as a near-duplicate of document $D_2$ if more than 90% of the words in the documents were the same.

There are two scenarios for near-duplicate detection. One is the *search* scenario, where the goal is to find near-duplicates of a given document $D$. This, like all search problems, conceptually involves the comparison of the query document to all other documents. For a collection containing $N$ documents, the number of comparisons required will be $\mathcal{O}(N)$. The other scenario, *discovery*, involves finding all pairs of near-duplicate documents in the collection. This process requires $\mathcal{O}(N^2)$ comparisons. Although information retrieval techniques that measure similarity using word-based representations of documents have been shown to be effective for identifying near-duplicates in the search scenario, the computational requirements of the discovery scenario have meant that new techniques have been developed for deriving compact representations of documents. These compact representations are known as *fingerprints*.

The basic process of generating fingerprints is as follows:

1. The document is parsed into words. Non-word content, such as punctuation, HTML tags, and additional whitespace, is removed (see section 4.3).
2. The words are grouped into contiguous *n-grams* for some $n$. These are usually overlapping sequences of words (see section 4.3.5), although some techniques use non-overlapping sequences.
3. Some of the n-grams are selected to represent the document.
4. The selected n-grams are hashed to improve retrieval efficiency and further reduce the size of the representation.
5. The hash values are stored, typically in an inverted index.

There are a number of fingerprinting algorithms that use this general approach, and they differ mainly in how subsets of the n-grams are selected. Selecting a fixed number of n-grams at random does not lead to good performance in terms of finding near-duplicates. Consider two near-identical documents, $D_1$ and $D_2$. The fingerprints generated from n-grams selected randomly from document $D_1$ are unlikely to have a high overlap with the fingerprints generated from a different set of n-grams selected randomly from $D_2$. A more effective technique uses prespecified combinations of characters, and selects n-grams that begin with those characters. Another popular technique, called *0 mod p*, is to select all n-grams whose hash value *modulo* $p$ is zero, where $p$ is a parameter.

Figure 3.14 illustrates the fingerprinting process using overlapping 3-grams, hypothetical hash values, and the *0 mod p* selection method with a $p$ value of 4. Note that after the selection process, the document (or sentence in this case) is represented by fingerprints for the n-grams "fish include fish", "found in tropical",

Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

(a) Original text

tropical fish include, fish include fish, include fish found, fish found in, found in tropical, in tropical environments, tropical environments around, environments around the, around the world, the world including, world including both, including both freshwater, both freshwater and, freshwater and salt, and salt water, salt water species

(b) 3-grams

938 664 463 822 492 798 78 969 143 236 913 908 694 553 870 779

(c) Hash values

664 492 236 908

(d) Selected hash values using *0 mod 4*

**Fig. 3.14.** Example of fingerprinting process

"the world including", and "including both freshwater". In large-scale applications, such as finding near-duplicates on the Web, the n-grams are typically 5–10 words long and the hash values are 64 bits.[3]

Near-duplicate documents are found by comparing the fingerprints that represent them. Near-duplicate pairs are defined by the number of shared fingerprints or the ratio of shared fingerprints to the total number of fingerprints used to represent the pair of documents. Fingerprints do not capture all of the information in the document, however, and consequently this leads to errors in the detection of near-duplicates. Appropriate selection techniques can reduce these errors, but not eliminate them. As we mentioned, evaluations have shown that comparing word-based representations using a similarity measure such as the cosine correlation (see section 7.1.2) is generally significantly more effective than fingerprinting methods for finding near-duplicates. The problem with these methods is their efficiency.

[3] The hash values are usually generated using *Rabin fingerprinting* (Broder et al., 1997), named after the Israeli computer scientist Michael Rabin.

A recently developed fingerprinting technique called `simhash` (Charikar, 2002) combines the advantages of the word-based similarity measures with the efficiency of fingerprints based on hashing. It has the unusual property for a hashing function that similar documents have similar hash values. More precisely, the similarity of two pages as measured by the cosine correlation measure is proportional to the number of bits that are the same in the fingerprints generated by `simhash`.

The procedure for calculating a `simhash` fingerprint is as follows:

1. Process the document into a set of features with associated weights. We will assume the simple case where the features are words weighted by their frequency. Other weighting schemes are discussed in Chapter 7.
2. Generate a hash value with $b$ bits (the desired size of the fingerprint) for each word. The hash value should be unique for each word.
3. In b-dimensional vector $V$, update the components of the vector by adding the weight for a word to every component for which the corresponding bit in the word's hash value is 1, and subtracting the weight if the value is 0.
4. After all words have been processed, generate a $b$-bit fingerprint by setting the $i$th bit to 1 if the $i$th component of $V$ is positive, or 0 otherwise.

Figure 3.15 shows an example of this process for an 8-bit fingerprint. Note that common words (stopwords) are removed as part of the text processing. In practice, much larger values of $b$ are used. Henzinger (2006) describes a large-scale Web-based evaluation where the fingerprints had 384 bits. A web page is defined as a near-duplicate of another page if the `simhash` fingerprints agree on more than 372 bits. This study showed significant effectiveness advantages for the `simhash` approach compared to fingerprints based on n-grams.

## 3.8 Removing Noise

Many web pages contain text, links, and pictures that are not directly related to the main content of the page. For example, Figure 3.16 shows a web page containing a news story. The main content of the page (the story) is outlined in black. This *content block* takes up less than 20% of the display area of the page, and the rest is made up of banners, advertisements, images, general navigation links, services (such as search and alerts), and miscellaneous information, such as copyright. From the perspective of the search engine, this additional material in the web page is mostly *noise* that could negatively affect the ranking of the page. A major component of the representation of a page used in a search engine is based

Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

(a) Original text

tropical 2 fish 2 include 1 found 1 environments 1 around 1 world 1 including 1 both 1 freshwater 1 salt 1 water 1 species 1

(b) Words with weights

| | | | | | |
|---|---|---|---|---|---|
| tropical | 01100001 | fish | 10101011 | include | 11100110 |
| found | 00011110 | environments | 00101101 | around | 10001011 |
| world | 00101010 | including | 11000000 | both | 10101110 |
| freshwater | 00111111 | salt | 10110101 | water | 00100101 |
| species | 11101110 | | | | |

(c) 8 bit hash values

1 -5 9 -9 3 1 3 3

(d) Vector $V$ formed by summing weights

1 0 1 0 1 1 1 1

(e) 8-bit fingerprint formed from $V$

**Fig. 3.15.** Example of `simhash` fingerprinting process

on word counts, and the presence of a large number of words unrelated to the main topic can be a problem. For this reason, techniques have been developed to detect the content blocks in a web page and either ignore the other material or reduce its importance in the indexing process.

Finn et al. (2001) describe a relatively simple technique based on the observation that there are less HTML tags in the text of the main content of typical web pages than there is in the additional material. Figure 3.17 (also known as a *document slope curve*) shows the cumulative distribution of tags in the example web page from Figure 3.16, as a function of the total number of tokens (words or other non-tag strings) in the page. The main text content of the page corresponds to the "plateau" in the middle of the distribution. This flat area is relatively small because of the large amount of formatting and presentation information in the HTML source for the page.


**Fig. 3.16.** Main content block in a web page

One way to detect the largest flat area of the distribution is to represent a web page as a sequence of bits, where $b_n = 1$ indicates that the $n$th token is a tag, and $b_n = 0$ otherwise. Certain tags that are mostly used to format text, such as font changes, headings, and table tags, are ignored (i.e., are represented by a 0 bit). The detection of the main content can then be viewed as an optimization problem where we find values of $i$ and $j$ to maximize both the number of tags below $i$ and above $j$ and the number of non-tag tokens between $i$ and $j$. This corresponds to maximizing the corresponding objective function:

$$\sum_{n=0}^{i-1} b_n + \sum_{n=i}^{j} (1 - b_n) + \sum_{n=j+1}^{N-1} b_n$$

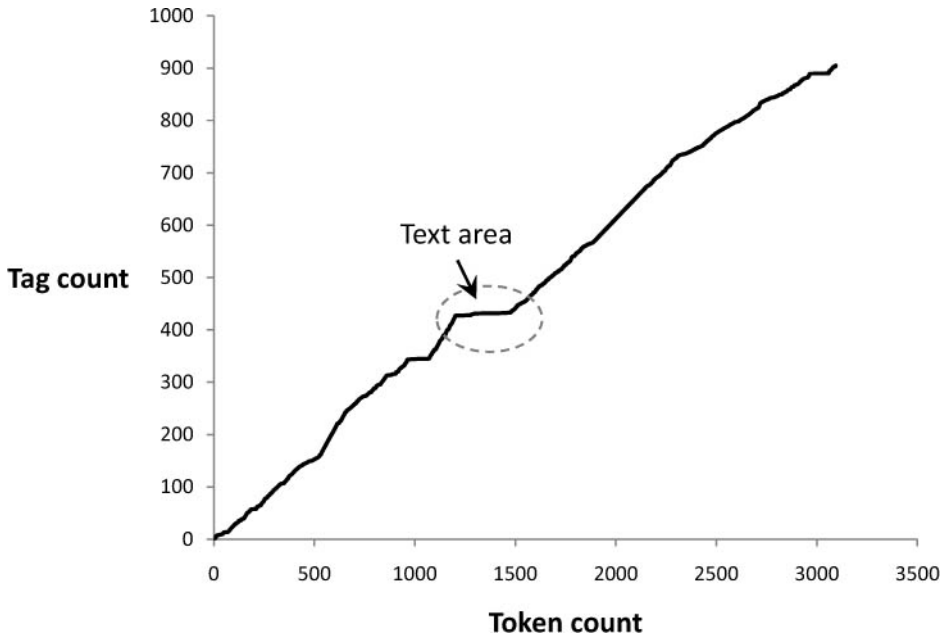


**Fig. 3.17.** Tag counts used to identify text blocks in a web page

where $N$ is the number of tokens in the page. This can be done simply by scanning the possible values for $i$ and $j$ and computing the objective function. Note that this procedure will only work when the proportion of text tokens in the non-content section is lower than the proportion of tags, which is not the case for the web page in Figure 3.17. Pinto et al. (2002) modified this approach to use a text window to search for low-slope sections of the document slope curve.

The structure of the web page can also be used more directly to identify the content blocks in the page. To display a web page using a browser, an HTML parser interprets the structure of the page specified using the tags, and creates a Document Object Model (DOM) representation. The tree-like structure represented by the DOM can be used to identify the major components of the web page. Figure 3.18 shows part of the DOM structure[4] for the example web page in Figure 3.16. The part of the structure that contains the text of the story is indicated by the comment cnnArticleContent. Gupta et al. (2003) describe an approach that

[4] This was generated using the DOM Inspector tool in the Firefox browser.

navigates the DOM tree recursively, using a variety of filtering techniques to remove and modify nodes in the tree and leave only content. HTML elements such as images and scripts are removed by simple filters. More complex filters remove advertisements, lists of links, and tables that do not have "substantive" content.

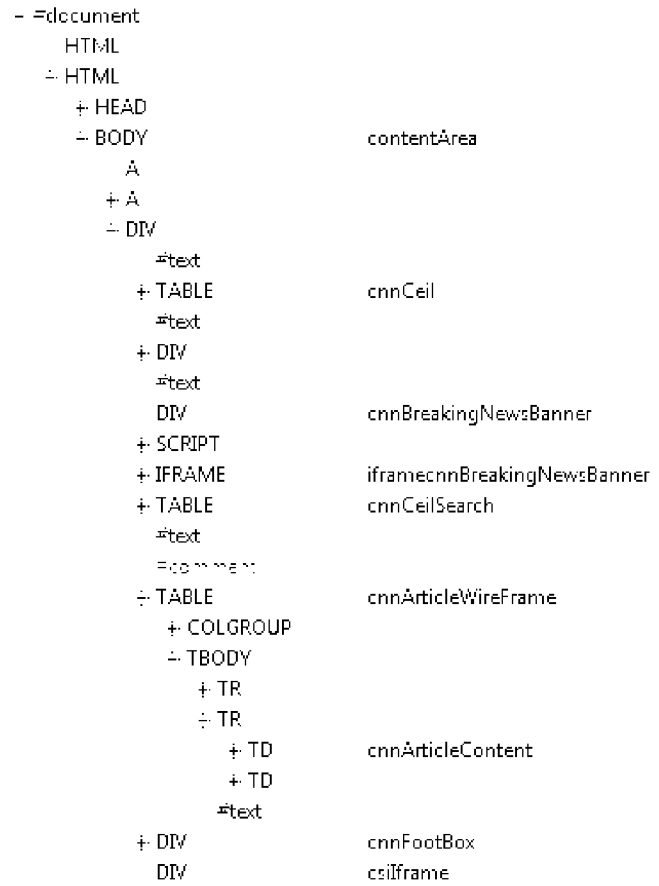


**Fig. 3.18.** Part of the DOM structure for the example web page

The DOM structure provides useful information about the components of a web page, but it is complex and is a mixture of logical and layout components. In Figure 3.18, for example, the content of the article is buried in a table cell (TD tag) in a row (TR tag) of an HTML table (TABLE tag). The table is being used in this case to specify layout rather than semantically related data. Another approach to

identifying the content blocks in a page focuses on the layout and presentation of the web page. In other words, visual features—such as the position of the block, the size of the font used, the background and font colors, and the presence of separators (such as lines and spaces)—are used to define blocks of information that would be apparent to the user in the displayed web page. Yu et al. (2003) describe an algorithm that constructs a hierarchy of visual blocks from the DOM tree and visual features.

The first algorithm we discussed, based on the distribution of tags, is quite effective for web pages with a single content block. Algorithms that use the DOM structure and visual analysis can deal with pages that may have several content blocks. In the case where there are several content blocks, the relative importance of each block can be used by the indexing process to produce a more effective representation. One approach to judging the importance of the blocks in a web page is to train a *classifier* that will assign an importance category based on visual and content features (R. Song et al., 2004).

## References and Further Reading

Cho and Garcia-Molina (2002, 2003) wrote a series of influential papers on web crawler design. Our discussion of page refresh policies is based heavily on Cho and Garcia-Molina (2003), and section 3.2.7 draws from Cho and Garcia-Molina (2002).

There are many open source web crawlers. The Heritrix crawler,[5] developed for the Internet Archive project, is a capable and scalable example. The system is developed in modules that are highly configurable at runtime, making it particularly suitable for experimentation.

Focused crawling attracted much attention in the early days of web search. Menczer and Belew (1998) and Chakrabarti et al. (1999) wrote two of the most influential papers. Menczer and Belew (1998) envision a focused crawler made of autonomous software agents, principally for a single user. The user enters a list of both URLs and keywords. The agent then attempts to find web pages that would be useful to the user, and the user can rate those pages to give feedback to the system. Chakrabarti et al. (1999) focus on crawling for specialized topical indexes. Their crawler uses a classifier to determine the topicality of crawled pages, as well as a distiller, which judges the quality of a page as a source of links to other topical

[5] http://crawler.archive.org

pages. They evaluate their system against a traditional, unfocused crawler to show that an unfocused crawler seeded with topical links is not sufficient to achieve a topical crawl. The broad link structure of the Web causes the unfocused crawler to quickly drift to other topics, while the focused crawler successfully stays on topic.

The Unicode specification is an incredibly detailed work, covering tens of thousands of characters (Unicode Consortium, 2006). Because of the nature of some non-Western scripts, many glyphs are formed from grouping a number of Unicode characters together, so the specification must detail not just what the characters are, but how they can be joined together. Characters are still being added to Unicode periodically.

Bergman (2001) is an extensive study of the deep Web. Even though this study is old by web standards, it shows how sampling through search engines can be used to help estimate the amount of unindexed content on the Web. This study estimated that 550 billion web pages existed in the deep Web, compared to 1 billion in the accessible Web. He et al. (2007) describe a more recent survey that shows that the deep Web has continued to expand rapidly in recent years. An example of a technique for generating searchable representations of deep Web databases, called query probing, is described by Ipeirotis and Gravano (2004).

Sitemaps, robots.txt files, RSS feeds, and Atom feeds each have their own specifications, which are available on the Web.[6] These formats show that successful web standards are often quite simple.

As we mentioned, database systems can be used to store documents from a web crawl for some applications. Our discussion of database systems was, however, limited mostly to a comparison with BigTable. There are a number of textbooks, such as Garcia-Molina et al. (2008), that provide much more information on how databases work, including details about important features such as query languages, locking, and recovery. BigTable, which we referenced frequently, was described in Chang et al. (2006). Other large Internet companies have built their own database systems with similar goals: large-scale distribution and high throughput, but without an expressive query language or detailed transaction support. The Dynamo system from Amazon has low latency guarantees (DeCandia et al., 2007), and Yahoo! uses their UDB system to store large datasets (Baeza-Yates & Ramakrishnan, 2008).

[6] http://www.sitemaps.org
http://www.robotstxt.org
http://www.rssboard.org/rss-specification
http://www.rfc-editor.org/rfc/rfc5023.txt

We mentioned DEFLATE (Deutsch, 1996) and LZW (Welch, 1984) as specific document compression algorithms in the text. DEFLATE is the basis for the popular Zip, gzip, and zlib compression tools. LZW is the basis of the Unix `compress` command, and is also found in file formats such as GIF, PostScript, and PDF. The text by Witten et al. (1999) provides detailed discussions about text and image compression algorithms.

Hoad and Zobel (2003) provide both a review of fingerprinting techniques and a comparison to word-based similarity measures for near-duplicate detection. Their evaluation focused on finding versions of documents and plagiarized documents. Bernstein and Zobel (2006) describe a technique for using full fingerprinting (no selection) for the task of finding *co-derivatives*, which are documents derived from the same source. Bernstein and Zobel (2005) examined the impact of duplication on evaluations of retrieval effectiveness. They showed that about 15% of the relevant documents for one of the TREC tracks were redundant, which could significantly affect the impact of the results from a user's perspective.

Henzinger (2006) describes a large-scale evaluation of near-duplicate detection on the Web. The two techniques compared were a version of Broder's "shingling" algorithm (Broder et al., 1997; Fetterly et al., 2003) and `simhash` (Charikar, 2002). Henzinger's study, which used 1.6 billion pages, showed that neither method worked well for detecting redundant documents *on the same site* because of the frequent use of "boilerplate" text that makes different pages look similar. For pages on different sites, the `simhash` algorithm achieved a precision of 50% (meaning that of those pages that were declared "near-duplicate" based on the similarity threshold, 50% were correct), whereas the Broder algorithm produced a precision of 38%.

A number of papers have been written about techniques for extracting content from web pages. Yu et al. (2003) and Gupta et al. (2003) are good sources for references to these papers.

## Exercises

**3.1.** Suppose you have two collections of documents. The smaller collection is full of useful, accurate, high-quality information. The larger collection contains a few high-quality documents, but also contains lower-quality text that is old, out-of-date, or poorly written. What are some reasons for building a search engine for only the small collection? What are some reasons for building a search engine that covers both collections?

**3.2.** Suppose you have a network connection that can transfer 10MB per second. If each web page is 10K and requires 500 milliseconds to transfer, how many threads does your web crawler need to fully utilize the network connection? If your crawler needs to wait 10 seconds between requests to the same web server, what is the minimum number of distinct web servers the system needs to contact each minute to keep the network connection fully utilized?

**3.3.** What is the advantage of using HEAD requests instead of GET requests during crawling? When would a crawler use a GET request instead of a HEAD request?

**3.4.** Why do crawlers not use POST requests?

**3.5.** Name the three types of sites mentioned in the chapter that compose the deep Web.

**3.6.** How would you design a system to automatically enter data into web forms in order to crawl deep Web pages? What measures would you use to make sure your crawler's actions were not destructive (for instance, so that it doesn't add random blog comments).

**3.7.** Write a program that can create a valid sitemap based on the contents of a directory on your computer's hard disk. Assume that the files are accessible from a website at the URL http://www.example.com. For instance, if there is a file in your directory called homework.pdf, this would be available at http://www.example.com/homework.pdf. Use the real modification date on the file as the last modified time in the sitemap, and to help estimate the change frequency.

**3.8.** Suppose that, in an effort to crawl web pages faster, you set up two crawling machines with different starting seed URLs. Is this an effective strategy for distributed crawling? Why or why not?

**3.9.** Write a simple single-threaded web crawler. Starting from a single input URL (perhaps a professor's web page), the crawler should download a page and then wait at least five seconds before downloading the next page. Your program should find other pages to crawl by parsing link tags found in previously crawled documents.

**3.10.** UTF-16 is used in Java and Windows®. Compare it to UTF-8.

**3.11.** How does BigTable handle hardware failure?

**3.12.** Design a compression algorithm that compresses HTML tags. Your algorithm should detect tags in an HTML file and replace them with a code of your own design that is smaller than the tag itself. Write an encoder and decoder program.

**3.13.** Generate checksums for a document by adding the bytes of the document and by using the Unix command `cksum`. Edit the document and see if both checksums change. Can you change the document so that the simple checksum does not change?

**3.14.** Write a program to generate `simhash` fingerprints for documents. You can use any reasonable hash function for the words. Use the program to detect duplicates on your home computer. Report on the accuracy of the detection. How does the detection accuracy vary with fingerprint size?

**3.15.** Plot the document slope curves for a sample of web pages. The sample should include at least one page containing a news article. Test the accuracy of the simple optimization algorithm for detecting the main content block. Write your own program or use the code from http://www.aidanf.net/software/bte-body-text-extraction. Describe the cases where the algorithm fails. Would an algorithm that searched explicitly for low-slope areas of the document slope curve be successful in these cases?

**3.16.** Give a high-level outline of an algorithm that would use the DOM structure to identify content information in a web page. In particular, describe heuristics you would use to identify content and non-content elements of the structure.