20 *Web crawling and indexes*

20.1 Overview

Web crawling is the process by which we gather pages from the Web, in order to index them and support a search engine. The objective of crawling is to quickly and efficiently gather as many useful web pages as possible, together with the link structure that interconnects them. In Chapter 19 we studied the complexities of the Web stemming from its creation by millions of uncoordinated individuals. In this chapter we study the resulting difficulties for crawling the Web. The focus of this chapter is the component shown in Figure 19.7 as *web crawler*; it is sometimes referred to as a *spider*.

WEB CRAWLER
SPIDER

The goal of this chapter is not to describe how to build the crawler for a full-scale commercial web search engine. We focus instead on a range of issues that are generic to crawling from the student project scale to substantial research projects. We begin (Section 20.1.1) by listing desiderata for web crawlers, and then discuss in Section 20.2 how each of these issues is addressed. The remainder of this chapter describes the architecture and some implementation details for a distributed web crawler that satisfies these features. Section 20.3 discusses distributing indexes across many machines for a web-scale implementation.

20.1.1 Features a crawler *must* provide

We list the desiderata for web crawlers in two categories: features that web crawlers *must* provide, followed by features they *should* provide.

Robustness: The Web contains servers that create *spider traps*, which are generators of web pages that mislead crawlers into getting stuck fetching an infinite number of pages in a particular domain. Crawlers must be designed to be resilient to such traps. Not all such traps are malicious; some are the inadvertent side-effect of faulty website development.

Politeness: Web servers have both implicit and explicit policies regulating the rate at which a crawler can visit them. These politeness policies must be respected.

20.1.2 Features a crawler *should* provide

Distributed: The crawler should have the ability to execute in a distributed fashion across multiple machines.

Scalable: The crawler architecture should permit scaling up the crawl rate by adding extra machines and bandwidth.

Performance and efficiency: The crawl system should make efficient use of various system resources including processor, storage and network bandwidth.

Quality: Given that a significant fraction of all web pages are of poor utility for serving user query needs, the crawler should be biased towards fetching "useful" pages first.

Freshness: In many applications, the crawler should operate in continuous mode: it should obtain fresh copies of previously fetched pages. A search engine crawler, for instance, can thus ensure that the search engine's index contains a fairly current representation of each indexed web page. For such continuous crawling, a crawler should be able to crawl a page with a frequency that approximates the rate of change of that page.

Extensible: Crawlers should be designed to be extensible in many ways – to cope with new data formats, new fetch protocols, and so on. This demands that the crawler architecture be modular.

20.2 Crawling

The basic operation of any hypertext crawler (whether for the Web, an intranet or other hypertext document collection) is as follows. The crawler begins with one or more URLs that constitute a *seed set*. It picks a URL from this seed set, then fetches the web page at that URL. The fetched page is then parsed, to extract both the text and the links from the page (each of which points to another URL). The extracted text is fed to a text indexer (described in Chapters 4 and 5). The extracted links (URLs) are then added to a *URL frontier*, which at all times consists of URLs whose corresponding pages have yet to be fetched by the crawler. Initially, the URL frontier contains the seed set; as pages are fetched, the corresponding URLs are deleted from the URL frontier. The entire process may be viewed as traversing the web graph (see

Chapter 19). In continuous crawling, the URL of a fetched page is added back to the frontier for fetching again in the future.

This seemingly simple recursive traversal of the web graph is complicated by the many demands on a practical web crawling system: the crawler has to be distributed, scalable, efficient, polite, robust and extensible while fetching pages of high quality. We examine the effects of each of these issues. Our treatment follows the design of the *Mercator* crawler that has formed the basis of a number of research and commercial crawlers. As a reference point, fetching a billion pages (a small fraction of the static Web at present) in a month-long crawl requires fetching several hundred pages each second. We will see how to use a multi-threaded design to address several bottlenecks in the overall crawler system in order to attain this fetch rate.

MERCATOR

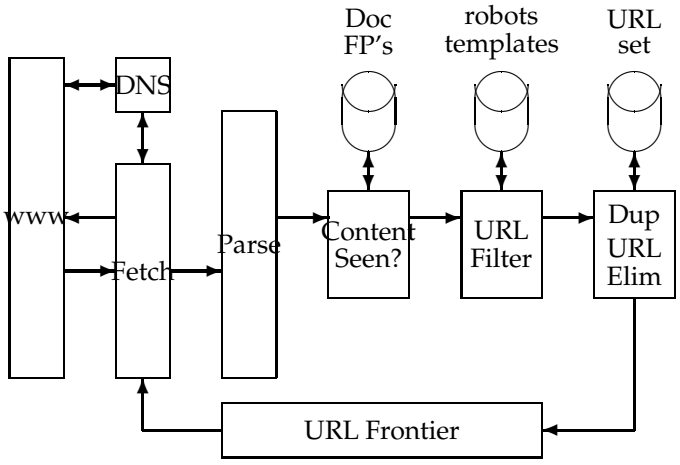
Before proceeding to this detailed description, we reiterate for readers who may attempt to build crawlers of some basic properties any non-professional crawler should satisfy:

1. Only one connection should be open to any given host at a time.
2. A waiting time of a few seconds should occur between successive requests to a host.
3. Politeness restrictions detailed in Section 20.2.1 should be obeyed.

20.2.1 Crawler architecture

The simple scheme outlined above for crawling demands several modules that fit together as shown in Figure 20.1.

1. The URL frontier, containing URLs yet to be fetched in the current crawl (in the case of continuous crawling, a URL may have been fetched previously but is back in the frontier for re-fetching). We describe this further in Section 20.2.3.
2. A *DNS resolution* module that determines the web server from which to fetch the page specified by a URL. We describe this further in Section 20.2.2.
3. A fetch module that uses the http protocol to retrieve the web page at a URL.
4. A parsing module that extracts the text and set of links from a fetched web page.
5. A duplicate elimination module that determines whether an extracted link is already in the URL frontier or has recently been fetched.



► **Figure 20.1** The basic crawler architecture.

Crawling is performed by anywhere from one to potentially hundreds of threads, each of which loops through the logical cycle in Figure 20.1. These threads may be run in a single process, or be partitioned amongst multiple processes running at different nodes of a distributed system. We begin by assuming that the URL frontier is in place and non-empty and defer our description of the implementation of the URL frontier to Section 20.2.3. We follow the progress of a single URL through the cycle of being fetched, passing through various checks and filters, then finally (for continuous crawling) being returned to the URL frontier.

A crawler thread begins by taking a URL from the frontier and fetching the web page at that URL, generally using the http protocol. The fetched page is then written into a temporary store, where a number of operations are performed on it. Next, the page is parsed and the text as well as the links in it are extracted. The text (with any tag information – e.g., terms in boldface) is passed on to the indexer. Link information including anchor text is also passed on to the indexer for use in ranking in ways that are described in Chapter 21. In addition, each extracted link goes through a series of tests to determine whether the link should be added to the URL frontier.

First, the thread tests whether a web page with the same content has already been seen at another URL. The simplest implementation for this would use a simple fingerprint such as a checksum (placed in a store labeled "Doc FP's" in Figure 20.1). A more sophisticated test would use shingles instead

of fingerprints, as described in Chapter 19.

Next, a *URL filter* is used to determine whether the extracted URL should be excluded from the frontier based on one of several tests. For instance, the crawl may seek to exclude certain domains (say, all .com URLs) – in this case the test would simply filter out the URL if it were from the .com domain. A similar test could be inclusive rather than exclusive. Many hosts on the Web place certain portions of their websites off-limits to crawling, under a standard known as the *Robots Exclusion Protocol*. This is done by placing a file with the name robots.txt at the root of the URL hierarchy at the site. Here is an example robots.txt file that specifies that no robot should visit any URL whose position in the file hierarchy starts with /yoursite/temp/, except for the robot called "searchengine".

ROBOTS EXCLUSION PROTOCOL

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

User-agent: searchengine
Disallow:
```

The robots.txt file must be fetched from a website in order to test whether the URL under consideration passes the robot restrictions, and can therefore be added to the URL frontier. Rather than fetch it afresh for testing on each URL to be added to the frontier, a cache can be used to obtain a recently fetched copy of the file for the host. This is especially important since many of the links extracted from a page fall within the host from which the page was fetched and therefore can be tested against the host's robots.txt file. Thus, by performing the filtering during the link extraction process, we would have especially high locality in the stream of hosts that we need to test for robots.txt files, leading to high cache hit rates. Unfortunately, this runs afoul of webmasters' politeness expectations. A URL (particularly one referring to a low-quality or rarely changing document) may be in the frontier for days or even weeks. If we were to perform the robots filtering *before* adding such a URL to the frontier, its robots.txt file could have changed by the time the URL is dequeued from the frontier and fetched. We must consequently perform robots-filtering immediately before attempting to fetch a web page. As it turns out, maintaining a cache of robots.txt files is still highly effective; there is sufficient locality even in the stream of URLs dequeued from the URL frontier.

URL NORMALIZATION

Next, a URL should be *normalized* in the following sense: often the HTML encoding of a link from a web page p indicates the target of that link relative to the page p. Thus, there is a relative link encoded thus in the HTML of the page en.wikipedia.org/wiki/Main_Page:

```
<a href="/wiki/Wikipedia:General_disclaimer" title="Wikipedia:General
disclaimer">Disclaimers</a>
```

points to the URL http://en.wikipedia.org/wiki/Wikipedia:General_disclaimer.

Finally, the URL is checked for duplicate elimination: if the URL is already in the frontier or (in the case of a non-continuous crawl) already crawled, we do not add it to the frontier. When the URL is added to the frontier, it is assigned a priority based on which it is eventually removed from the frontier for fetching. The details of this priority queuing are in Section 20.2.3.

Certain housekeeping tasks are typically performed by a dedicated thread. This thread is generally quiescent except that it wakes up once every few seconds to log crawl progress statistics (URLs crawled, frontier size, etc.), decide whether to terminate the crawl, or (once every few hours of crawling) checkpoint the crawl. In checkpointing, a snapshot of the crawler's state (say, the URL frontier) is committed to disk. In the event of a catastrophic crawler failure, the crawl is restarted from the most recent checkpoint.

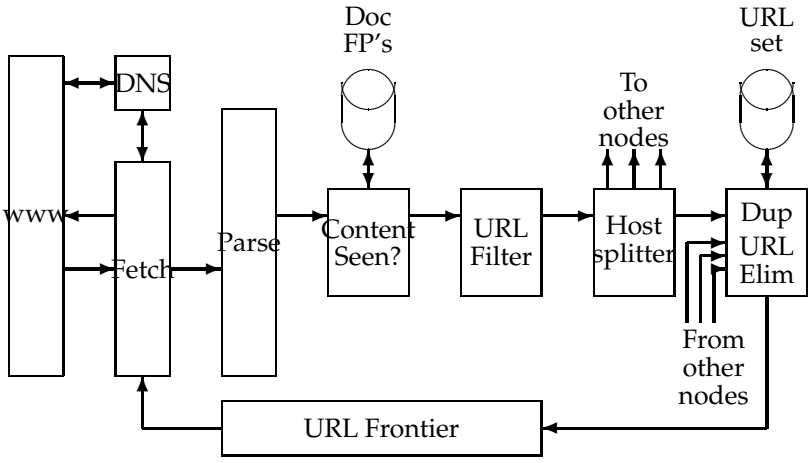
Distributing the crawler

We have mentioned that the threads in a crawler could run under different processes, each at a different node of a distributed crawling system. Such distribution is essential for scaling; it can also be of use in a geographically distributed crawler system where each node crawls hosts "near" it. Partitioning the hosts being crawled amongst the crawler nodes can be done by a hash function, or by some more specifically tailored policy. For instance, we may locate a crawler node in Europe to focus on European domains, although this is not dependable for several reasons – the routes that packets take through the internet do not always reflect geographic proximity, and in any case the domain of a host does not always reflect its physical location.

How do the various nodes of a distributed crawler communicate and share URLs? The idea is to replicate the flow of Figure 20.1 at each node, with one essential difference: following the URL filter, we use a *host splitter* to dispatch each surviving URL to the crawler node responsible for the URL; thus the set of hosts being crawled is partitioned among the nodes. This modified flow is shown in Figure 20.2. The output of the host splitter goes into the Duplicate URL Eliminator block of each other node in the distributed system.

The "Content Seen?" module in the distributed architecture of Figure 20.2 is, however, complicated by several factors:

1. Unlike the URL frontier and the duplicate elimination module, document fingerprints/shingles cannot be partitioned based on host name. There is nothing preventing the same (or highly similar) content from appearing on different web servers. Consequently, the set of fingerprints/shingles must be partitioned across the nodes based on some property of the fin-



► **Figure 20.2** Distributing the basic crawl architecture.

gerprint/shingle (say by taking the fingerprint modulo the number of nodes). The result of this locality-mismatch is that most "Content Seen?" tests result in a remote procedure call (although it is possible to batch lookup requests).

2. There is very little locality in the stream of document fingerprints/shingles. Thus, caching popular fingerprints does not help (since there are no popular fingerprints).

3. Documents change over time and so, in the context of continuous crawling, we must be able to delete their outdated fingerprints/shingles from the content-seen set(s). In order to do so, it is necessary to save the fingerprint/shingle of the document in the URL frontier, along with the URL itself.

20.2.2 DNS resolution

IP ADDRESS

Each web server (and indeed any host connected to the internet) has a unique *IP address*: a sequence of four bytes generally represented as four integers separated by dots; for instance 207.142.131.248 is the numerical IP address associated with the host www.wikipedia.org. Given a URL such as www.wikipedia.org in textual form, translating it to an IP address (in this case, 207.142.131.248) is

DNS RESOLUTION

a process known as *DNS resolution* or DNS lookup; here DNS stands for *Domain Name Service*. During DNS resolution, the program that wishes to perform this translation (in our case, a component of the web crawler) contacts a

DNS SERVER

DNS server that returns the translated IP address. (In practice the entire translation may not occur at a single DNS server; rather, the DNS server contacted initially may recursively call upon other DNS servers to complete the translation.) For a more complex URL such as en.wikipedia.org/wiki/Domain_Name_System, the crawler component responsible for DNS resolution extracts the host name – in this case en.wikipedia.org – and looks up the IP address for the host en.wikipedia.org.

DNS resolution is a well-known bottleneck in web crawling. Due to the distributed nature of the Domain Name Service, DNS resolution may entail multiple requests and round-trips across the internet, requiring seconds and sometimes even longer. Right away, this puts in jeopardy our goal of fetching several hundred documents a second. A standard remedy is to introduce caching: URLs for which we have recently performed DNS lookups are likely to be found in the DNS cache, avoiding the need to go to the DNS servers on the internet. However, obeying politeness constraints (see Section 20.2.3) limits the of cache hit rate.

There is another important difficulty in DNS resolution; the lookup implementations in standard libraries (likely to be used by anyone developing a crawler) are generally synchronous. This means that once a request is made to the Domain Name Service, other crawler threads at that node are blocked until the first request is completed. To circumvent this, most web crawlers implement their own DNS resolver as a component of the crawler. Thread i executing the resolver code sends a message to the DNS server and then performs a timed wait: it resumes either when being signaled by another thread or when a set time quantum expires. A single, separate DNS thread listens on the standard DNS port (port 53) for incoming response packets from the name service. Upon receiving a response, it signals the appropriate crawler thread (in this case, i) and hands it the response packet if i has not yet resumed because its time quantum has expired. A crawler thread that resumes because its wait time quantum has expired retries for a fixed number of attempts, sending out a new message to the DNS server and performing a timed wait each time; the designers of Mercator recommend of the order of five attempts. The time quantum of the wait increases exponentially with each of these attempts; Mercator started with one second and ended with roughly 90 seconds, in consideration of the fact that there are host names that take tens of seconds to resolve.

20.2.3 The URL frontier

The URL frontier at a node is given a URL by its crawl process (or by the host splitter of another crawl process). It maintains the URLs in the frontier and regurgitates them in some order whenever a crawler thread seeks a URL. Two important considerations govern the order in which URLs are returned by the frontier. First, high-quality pages that change frequently should be prioritized for frequent crawling. Thus, the priority of a page should be a function of both its change rate and its quality (using some reasonable quality estimate). The combination is necessary because a large number of spam pages change completely on every fetch.

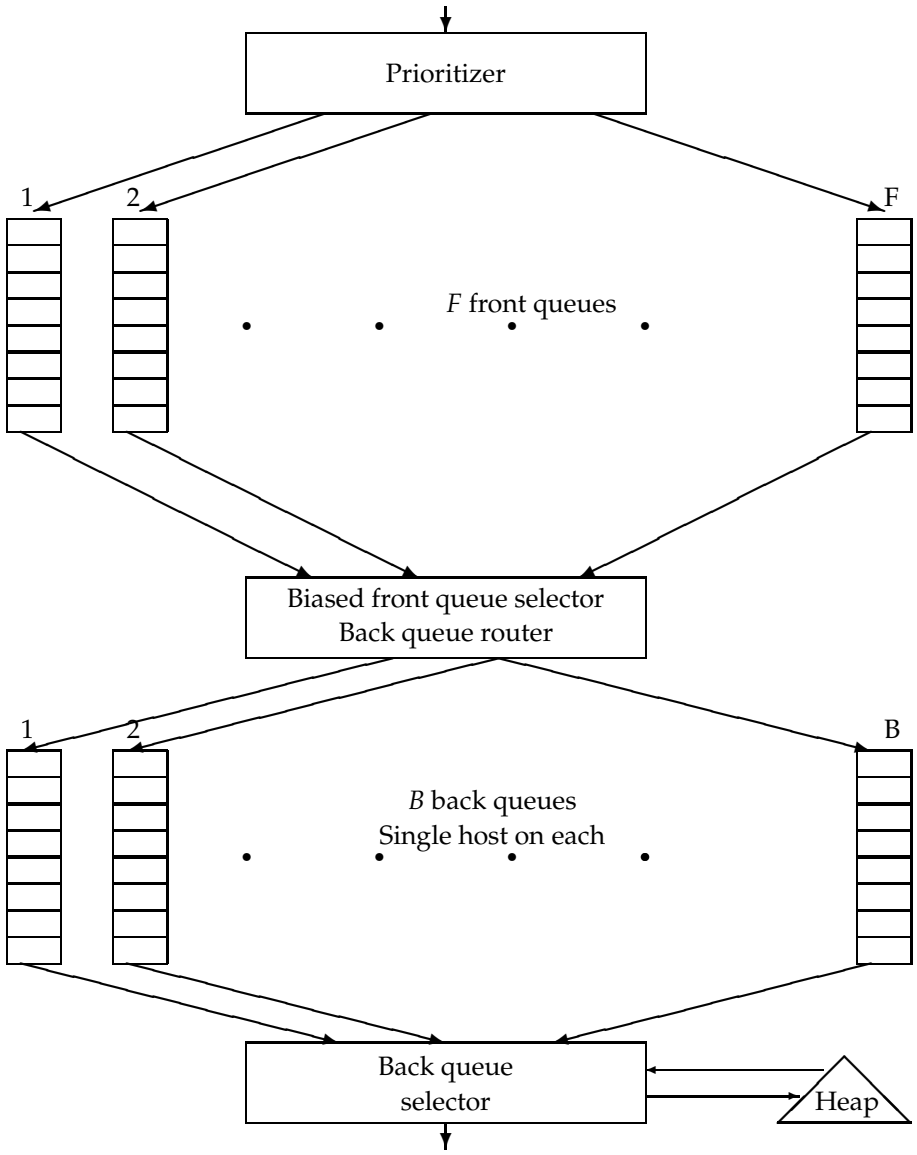
The second consideration is politeness: we must avoid repeated fetch requests to a host within a short time span. The likelihood of this is exacerbated because of a form of locality of reference: many URLs link to other URLs at the same host. As a result, a URL frontier implemented as a simple priority queue might result in a burst of fetch requests to a host. This might occur even if we were to constrain the crawler so that at most one thread could fetch from any single host at any time. A common heuristic is to insert a gap between successive fetch requests to a host that is an order of magnitude larger than the time taken for the most recent fetch from that host.

Figure 20.3 shows a polite and prioritizing implementation of a URL frontier. Its goals are to ensure that (i) only one connection is open at a time to any host; (ii) a waiting time of a few seconds occurs between successive requests to a host and (iii) high-priority pages are crawled preferentially.

The two major sub-modules are a set of F *front queues* in the upper portion of the figure, and a set of B *back queues* in the lower part; all of these are FIFO queues. The front queues implement the prioritization, while the back queues implement politeness. In the flow of a URL added to the frontier as it makes its way through the front and back queues, a *prioritizer* first assigns to the URL an integer priority i between 1 and F based on its fetch history (taking into account the rate at which the web page at this URL has changed between previous crawls). For instance, a document that has exhibited frequent change would be assigned a higher priority. Other heuristics could be application-dependent and explicit – for instance, URLs from news services may always be assigned the highest priority. Now that it has been assigned priority i, the URL is now appended to the ith of the front queues.

Each of the B back queues maintains the following invariants: (i) it is non-empty while the crawl is in progress and (ii) it only contains URLs from a single host[1]. An auxiliary table T (Figure 20.4) is used to maintain the mapping from hosts to back queues. Whenever a back-queue is empty and is being re-filled from a front-queue, table T must be updated accordingly.

1. The number of hosts is assumed to far exceed B.



► **Figure 20.3** The URL frontier. URLs extracted from already crawled pages flow in at the top of the figure. A crawl thread requesting a URL extracts it from the bottom of the figure. En route, a URL flows through one of several *front queues* that manage its priority for crawling, followed by one of several *back queues* that manage the crawler's politeness.

Host	Back queue
stanford.edu	23
microsoft.com	47
acm.org	12

► **Figure 20.4** Example of an auxiliary hosts-to-back queues table.

In addition, we maintain a heap with one entry for each back queue, the entry being the earliest time t_e at which the host corresponding to that queue can be contacted again.

A crawler thread requesting a URL from the frontier extracts the root of this heap and (if necessary) waits until the corresponding time entry t_e. It then takes the URL u at the head of the back queue j corresponding to the extracted heap root, and proceeds to fetch the URL u. After fetching u, the calling thread checks whether j is empty. If so, it picks a front queue and extracts from its head a URL v. The choice of front queue is biased (usually by a random process) towards queues of higher priority, ensuring that URLs of high priority flow more quickly into the back queues. We examine v to check whether there is already a back queue holding URLs from its host. If so, v is added to that queue and we reach back to the front queues to find another candidate URL for insertion into the now-empty queue j. This process continues until j is non-empty again. In any case, the thread inserts a heap entry for j with a new earliest time t_e based on the properties of the URL in j that was last fetched (such as when its host was last contacted as well as the time taken for the last fetch), then continues with its processing. For instance, the new entry t_e could be the current time plus ten times the last fetch time.

The number of front queues, together with the policy of assigning priorities and picking queues, determines the priority properties we wish to build into the system. The number of back queues governs the extent to which we can keep all crawl threads busy while respecting politeness. The designers of Mercator recommend a rough rule of three times as many back queues as crawler threads.

On a Web-scale crawl, the URL frontier may grow to the point where it demands more memory at a node than is available. The solution is to let most of the URL frontier reside on disk. A portion of each queue is kept in memory, with more brought in from disk as it is drained in memory.

? **Exercise 20.1**

Why is it better to partition hosts (rather than individual URLs) between the nodes of a distributed crawl system?

Exercise 20.2

Why should the host splitter precede the Duplicate URL Eliminator?

Exercise 20.3 [⋆⋆⋆]

In the preceding discussion we encountered two recommended "hard constants" – the increment on t_e being `ten` times the last fetch time, and the number of back queues being `three` times the number of crawl threads. How are these two constants related?

20.3 Distributing indexes

In Section 4.4 we described distributed indexing. We now consider the distribution of the index across a large computer cluster[2] that supports querying.

TERM PARTITIONING
DOCUMENT PARTITIONING

Two obvious alternative index implementations suggest themselves: *partitioning by terms*, also known as global index organization, and *partitioning by documents*, also know as local index organization. In the former, the dictionary of index terms is partitioned into subsets, each subset residing at a node. Along with the terms at a node, we keep the postings for those terms. A query is routed to the nodes corresponding to its query terms. In principle, this allows greater concurrency since a stream of queries with different query terms would hit different sets of machines.

In practice, partitioning indexes by vocabulary terms turns out to be non-trivial. Multi-word queries require the sending of long postings lists between sets of nodes for merging, and the cost of this can outweigh the greater concurrency. Load balancing the partition is governed not by an a priori analysis of relative term frequencies, but rather by the distribution of query terms and their co-occurrences, which can drift with time or exhibit sudden bursts. Achieving good partitions is a function of the co-occurrences of query terms and entails the clustering of terms to optimize objectives that are not easy to quantify. Finally, this strategy makes implementation of dynamic indexing more difficult.

A more common implementation is to partition by documents: each node contains the index for a subset of all documents. Each query is distributed to all nodes, with the results from various nodes being merged before presentation to the user. This strategy trades more local disk seeks for less inter-node communication. One difficulty in this approach is that global statistics used in scoring – such as idf – must be computed across the entire document collection even though the index at any single node only contains a subset of the documents. These are computed by distributed "background" processes that periodically refresh the node indexes with fresh global statistics.

How do we decide the partition of documents to nodes? Based on our development of the crawler architecture in Section 20.2.1, one simple approach would be to assign all pages from a host to a single node. This partitioning

2. Please note the different usage of "clusters" elsewhere in this book, in the sense of Chapters 16 and 17.

could follow the partitioning of hosts to crawler nodes. A danger of such partitioning is that on many queries, a preponderance of the results would come from documents at a small number of hosts (and hence a small number of index nodes).

A hash of each URL into the space of index nodes results in a more uniform distribution of query-time computation across nodes. At query time, the query is broadcast to each of the nodes, with the top k results from each node being merged to find the top k documents for the query. A common implementation heuristic is to partition the document collection into indexes of documents that are more likely to score highly on most queries (using, for instance, techniques in Chapter 21) and low-scoring indexes with the remaining documents. We only search the low-scoring indexes when there are too few matches in the high-scoring indexes, as described in Section 7.2.1.

20.4 Connectivity servers

CONNECTIVITY SERVER
CONNECTIVITY QUERIES

For reasons to become clearer in Chapter 21, web search engines require a *connectivity server* that supports fast *connectivity queries* on the web graph. Typical connectivity queries are *which URLs link to a given URL?* and *which URLs does a given URL link to?* To this end, we wish to store mappings in memory from URL to out-links, and from URL to in-links. Applications include crawl control, web graph analysis, sophisticated crawl optimization and *link analysis* (to be covered in Chapter 21).

Suppose that the Web had four billion pages, each with ten links to other pages. In the simplest form, we would require 32 bits or 4 bytes to specify each end (source and destination) of each link, requiring a total of

$$4 \times 10^9 \times 10 \times 8 = 3.2 \times 10^{11}$$

bytes of memory. Some basic properties of the web graph can be exploited to use well under 10% of this memory requirement. At first sight, we appear to have a data compression problem – which is amenable to a variety of standard solutions. However, our goal is not to simply compress the web graph to fit into memory; we must do so in a way that efficiently supports connectivity queries; this challenge is reminiscent of index compression (Chapter 5).

We assume that each web page is represented by a unique integer; the specific scheme used to assign these integers is described below. We build an *adjacency table* that resembles an inverted index: it has a row for each web page, with the rows ordered by the corresponding integers. The row for any page p contains a sorted list of integers, each corresponding to a web page that links to p. This table permits us to respond to queries of the form *which pages link to p?* In similar fashion we build a table whose entries are the pages linked to by p.

```
1: www.stanford.edu/alchemy
2: www.stanford.edu/biology
3: www.stanford.edu/biology/plant
4: www.stanford.edu/biology/plant/copyright
5: www.stanford.edu/biology/plant/people
6: www.stanford.edu/chemistry
```

► **Figure 20.5** A lexicographically ordered set of URLs.

This table representation cuts the space taken by the naive representation (in which we explicitly represent each link by its two end points, each a 32-bit integer) by 50%. Our description below will focus on the table for the links *from* each page; it should be clear that the techniques apply just as well to the table of links to each page. To further reduce the storage for the table, we exploit several ideas:

1. Similarity between lists: Many rows of the table have many entries in common. Thus, if we explicitly represent a prototype row for several similar rows, the remainder can be succinctly expressed in terms of the prototypical row.

2. Locality: many links from a page go to "nearby" pages – pages on the same host, for instance. This suggests that in encoding the destination of a link, we can often use small integers and thereby save space.

3. We use gap encodings in sorted lists: rather than store the destination of each link, we store the offset from the previous entry in the row.

We now develop each of these techniques.

In a *lexicographic* ordering of all URLs, we treat each URL as an alphanumeric string and sort these strings. Figure 20.5 shows a segment of this sorted order. For a true lexicographic sort of web pages, the domain name part of the URL should be inverted, so that www.stanford.edu becomes edu.stanford.www, but this is not necessary here since we are mainly concerned with links local to a single host.

To each URL, we assign its position in this ordering as the unique identifying integer. Figure 20.6 shows an example of such a numbering and the resulting table. In this example sequence, `www.stanford.edu/biology` is assigned the integer 2 since it is second in the sequence.

We next exploit a property that stems from the way most websites are structured to get similarity and locality. Most websites have a template with a set of links from each page in the site to a fixed set of pages on the site (such

```
1: 1, 2, 4, 8, 16, 32, 64
2: 1, 4, 9, 16, 25, 36, 49, 64
3: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
4: 1, 4, 8, 16, 25, 36, 49, 64
```

► **Figure 20.6** A four-row segment of the table of links.

as its copyright notice, terms of use, and so on). In this case, the rows corresponding to pages in a website will have many table entries in common. Moreover, under the lexicographic ordering of URLs, it is very likely that the pages from a website appear as contiguous rows in the table.

We adopt the following strategy: we walk down the table, encoding each table row in terms of the seven preceding rows. In the example of Figure 20.6, we could encode the fourth row as "the same as the row at offset 2 (meaning, two rows earlier in the table), with 9 replaced by 8". This requires the specification of the offset, the integer(s) dropped (in this case 9) and the integer(s) added (in this case 8). The use of only the seven preceding rows has two advantages: (i) the offset can be expressed with only 3 bits; this choice is optimized empirically (the reason for seven and not eight preceding rows is the subject of Exercise 20.4) and (ii) fixing the maximum offset to a small value like seven avoids having to perform an expensive search among many candidate prototypes in terms of which to express the current row.

What if none of the preceding seven rows is a good prototype for expressing the current row? This would happen, for instance, at each boundary between different websites as we walk down the rows of the table. In this case we simply express the row as starting from the empty set and "adding in" each integer in that row. By using gap encodings to store the gaps (rather than the actual integers) in each row, and encoding these gaps tightly based on the distribution of their values, we obtain further space reduction. In experiments mentioned in Section 20.5, the series of techniques outlined here appears to use as few as 3 bits per link, on average – a dramatic reduction from the 64 required in the naive representation.

While these ideas give us a representation of sizable web graphs that comfortably fit in memory, we still need to support connectivity queries. What is entailed in retrieving from this representation the set of links from a page? First, we need an index lookup from (a hash of) the URL to its row number in the table. Next, we need to reconstruct these entries, which may be encoded in terms of entries in other rows. This entails following the offsets to reconstruct these other rows – a process that in principle could lead through many levels of indirection. In practice however, this does not happen very often. A heuristic for controlling this can be introduced into the construc-

tion of the table: when examining the preceding seven rows as candidates from which to model the current row, we demand a threshold of similarity between the current row and the candidate prototype. This threshold must be chosen with care. If the threshold is set too high, we seldom use prototypes and express many rows afresh. If the threshold is too low, most rows get expressed in terms of prototypes, so that at query time the reconstruction of a row leads to many levels of indirection through preceding prototypes.

?

Exercise 20.4

We noted that expressing a row in terms of one of seven preceding rows allowed us to use no more than three bits to specify which of the preceding rows we are using as prototype. Why seven and not eight preceding rows? *(Hint: consider the case when none of the preceding seven rows is a good prototype.)*

Exercise 20.5

We noted that for the scheme in Section 20.4, decoding the links incident on a URL could result in many levels of indirection. Construct an example in which the number of levels of indirection grows linearly with the number of URLs.

20.5 References and further reading

The first web crawler appears to be Matthew Gray's Wanderer, written in the spring of 1993. The Mercator crawler is due to Najork and Heydon (Najork and Heydon 2001; 2002); the treatment in this chapter follows their work. Other classic early descriptions of web crawling include Burner (1997), Brin and Page (1998), Cho et al. (1998) and the creators of the Webbase system at Stanford (Hirai et al. 2000). Cho and Garcia-Molina (2002) give a taxonomy and comparative study of different modes of communication between the nodes of a distributed crawler. The Robots Exclusion Protocol standard is described at http://www.robotstxt.org/wc/exclusion.html. Boldi et al. (2002) and Shkapenyuk and Suel (2002) provide more recent details of implementing large-scale distributed web crawlers.

Our discussion of DNS resolution (Section 20.2.2) uses the current convention for internet addresses, known as IPv4 (for Internet Protocol version 4) – each IP address is a sequence of four bytes. In the future, the convention for addresses (collectively known as the internet *address space*) is likely to use a new standard known as IPv6 (http://www.ipv6.org/).

Tomasic and Garcia-Molina (1993) and Jeong and Omiecinski (1995) are key early papers evaluating term partitioning versus document partitioning for distributed indexes. Document partitioning is found to be superior, at least when the distribution of terms is skewed, as it typically is in practice. This result has generally been confirmed in more recent work (MacFarlane et al. 2000). But the outcome depends on the details of the distributed system;

at least one thread of work has reached the opposite conclusion (Ribeiro-Neto and Barbosa 1998, Badue et al. 2001). Sornil (2001) argues for a partitioning scheme that is a hybrid between term and document partitioning. Barroso et al. (2003) describe the distribution methods used at Google. The first implementation of a connectivity server was described by Bharat et al. (1998). The scheme discussed in this chapter, currently believed to be the best published scheme (achieving as few as 3 bits per link for encoding), is described in a series of papers by Boldi and Vigna (2004a;b).