September 26, 2001

SRC Research Report 173

High-Performance Web Crawling

Marc Najork
Allan Heydon

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.compaq.com/SRC/

Compaq Systems Research Center

SRC's charter is to advance the state of the art in computer systems by doing basic and applied research in support of our company's business objectives. Our interests and projects span scalable systems (including hardware, networking, distributed systems, and programming-language technology), the Internet (including the Web, e-commerce, and information retrieval), and human/computer interaction (including user-interface technology, computer-based appliances, and mobile computing). SRC was established in 1984 by Digital Equipment Corporation.

We test the value of our ideas by building hardware and software prototypes and assessing their utility in realistic settings. Interesting systems are too complex to be evaluated solely in the abstract; practical use enables us to investigate their properties in depth. This experience is useful in the short term in refining our designs and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this approach, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical character. Some of that lies in established fields of theoretical computer science, such as the analysis of algorithms, computer-aided geometric design, security and cryptography, and formal specification and verification. Other work explores new ground motivated by problems that arise in our systems research.

We are strongly committed to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences, while our technical note series allows timely dissemination of recent research findings. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

High-Performance Web Crawling

Marc Najork
Allan Heydon

September 26, 2001

Publication History

A version of this report appeared in *Handbook of Massive Data Sets (edited by J. Abello, P. Pardalos, and M. Resende).* ©2001. Kluwer Academic Publishers, Inc. Republished by permission.

Allan Heydon is currently working at Model N, Inc. He can be reached by email at aheydon@modeln.com or at caheydon@yahoo.com.

Abstract

High-performance web crawlers are an important component of many web services. For example, search services use web crawlers to populate their indices, comparison shopping engines use them to collect product and pricing information from online vendors, and the Internet Archive uses them to record a history of the Internet. The design of a high-performance crawler poses many challenges, both technical and social, primarily due to the large scale of the web. The web crawler must be able to download pages at a very high rate, yet it must not overwhelm any particular web server. Moreover, it must maintain data structures far too large to fit in main memory, yet it must be able to access and update them efficiently. This chapter describes our experience building and operating such a high-performance crawler.

1 Introduction

A web crawler (also known as a web robot or spider) is a program for downloading web pages. Given a set s of "seed" Uniform Resource Locators (URLs), the crawler repeatedly removes one URL from s, downloads the corresponding page, extracts all the URLs contained in it, and adds any previously unknown URLs to s.

Although the web crawling algorithm is conceptually simple, designing a high-performance web crawler comparable to the ones used by the major search engines is a complex endeavor. All the challenges inherent in building such a high-performance crawler are ultimately due to the scale of the web. In order to crawl a billion pages in a month, a crawler must download about 400 pages every second. Moreover, the crawler must store several data structures (such as the set s of URLs remaining to be downloaded) that must all scale gracefully beyond the limits of main memory.

We have built just such a high-performance web crawler, called Mercator, which has the following characteristics:

Distributed. A Mercator crawl can be distributed in a symmetric fashion across multiple machines for better performance.

Scalable. Mercator is scalable in two respects. First, due to its distributed architecture, Mercator's performance can be scaled by adding extra machines to the crawling cluster. Second, Mercator has been designed to be able to cope with a rapidly growing web. In particular, its data structures use a bounded amount of main memory, regardless of the size of the web being crawled. This is achieved by storing the vast majority of data on disk.

High performance. During our most recent crawl, which ran on four Compaq DS20E 666 MHz Alpha servers and which saturated our 160 Mbit/sec Internet connection, Mercator downloaded about 50 million documents per day over a period of 17 days.

Polite. Despite the need for speed, anyone running a web crawler that overloads web servers soon learns that such behavior is considered unacceptable. At the very least, a web crawler should not attempt to download multiple pages from the same web server simultaneously; better, it should impose a limit on the portion of a web server's resources it consumes. Mercator can be configured to obey either of these politeness policies.

Continuous. There are many crawling applications (such as maintaining a fresh search engine index) where it is desirable to continuously refetch previously downloaded pages. This naturally raises the question of how to interleave the downloading of old pages with newly discovered ones. Mercator solves this problem by providing a priority-based mechanism for scheduling URL downloads.

Extensible. No two crawling tasks are the same. Ideally, a crawler should be designed in a modular way, where new functionality can be added by third parties. Mercator achieves this ideal through a component-based architecture. Each of Mercator's main components is specified by an abstract interface. We have written numerous implementations of each component, and third parties can write new implementations from scratch or extend ours through object-oriented subclassing. To configure Mercator for a particular crawling task, users supply a configuration file that causes the appropriate components to be loaded dynamically.

Portable. Mercator is written entirely in Java, and thus runs on any platform for which there exists a Java virtual machine. In particular, it is known to run on Windows NT, Linux, Tru64 Unix, Solaris, and AIX.

There is a natural tension between the high performance requirement on the one hand, and the scalability, politeness, extensibility, and portability requirements on the other. Simultaneously supporting all of these features is a significant design and engineering challenge.

This chapter describes Mercator's design and implementation, the lessons we've learned in the process of building it, and our experiences in performing large crawls.

2 A Survey of Web Crawlers

Web crawlers are almost as old as the web itself [16]. The first crawler, Matthew Gray's Wanderer, was written in the spring of 1993, roughly coinciding with the first release of NCSA Mosaic [9]. Several papers about web crawling were presented at the first two World Wide Web conferences [7, 18, 20]. However, at the time, the web was three to four orders of magnitude smaller than it is today, so those systems did not address the scaling problems inherent in a crawl of today's web.

Obviously, all of the popular search engines use crawlers that must scale up to substantial portions of the web. However, due to the competitive nature of the search engine business, the designs of these crawlers have not been publicly described. There are two notable exceptions: the Google crawler and the Internet Archive crawler. Unfortunately, the descriptions of these crawlers in the literature are too terse to enable reproducibility.

The original Google crawler [2] (developed at Stanford) consisted of five functional components running in different processes. A *URL server process* read URLs out of a file and forwarded them to multiple crawler processes. Each *crawler process* ran on a different machine, was single-threaded, and used asynchronous I/O to fetch data from up to 300 web servers in parallel. The crawlers transmitted downloaded pages to a single *StoreServer process*, which compressed the pages and stored them to disk. The pages were then read back from disk by an *indexer process*, which extracted links from HTML pages and saved them to a different disk file. A *URL resolver process* read the link file, derelativized the URLs contained therein, and saved the absolute URLs to the disk file that was read by the URL server. Typically, three to four crawler machines were used, so the entire system required between four and eight machines.

Research on web crawling continues at Stanford even after Google has been transformed into a commercial effort. The Stanford WebBase project has implemented a high-performance distributed crawler, capable of downloading 50 to 100 documents per second [14]. Cho and others have also developed models of document update frequencies to inform the download schedule of incremental crawlers [5].

The Internet Archive also used multiple machines to crawl the web [4, 15]. Each crawler process was assigned up to 64 sites to crawl, and no site was assigned to more than one crawler. Each single-threaded crawler process read a list of seed URLs for its assigned sites from disk into per-site queues, and then used asynchronous I/O to fetch pages from these queues in parallel. Once a page was downloaded, the crawler extracted the links contained in it. If a link referred to the site of the page it was contained in, it was added to the appropriate site queue; otherwise it was logged to disk. Periodically, a batch process merged these logged "cross-site" URLs into the site-specific seed sets, filtering out duplicates in the process.

The WebFountain crawler shares several of Mercator's characteristics: it is distributed, continuous (the authors use the term "incremental"), polite, and configurable [6]. Unfortunately, as of this writing, WebFountain is in the early stages of its development, and data about its performance is not yet available.

3 Mercator's Architecture

The basic algorithm executed by any web crawler takes a list of *seed* URLs as its input and repeatedly executes the following steps:

> Remove a URL from the URL list, determine the IP address of its host name, download the corresponding document, and extract any links

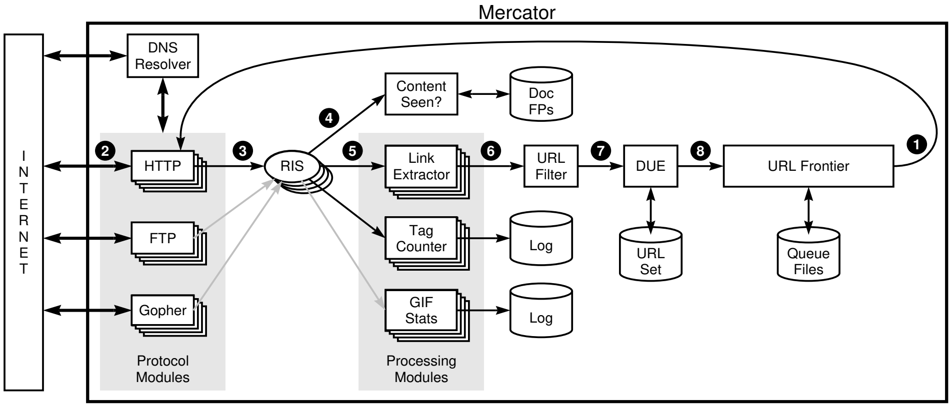


Figure 1: Mercator's main components.

contained in it. For each of the extracted links, ensure that it is an absolute URL (derelativizing it if necessary), and add it to the list of URLs to download, provided it has not been encountered before. If desired, process the downloaded document in other ways (e.g., index its content).

This basic algorithm requires a number of functional components:

- a component (called the *URL frontier*) for storing the list of URLs to download;
- a component for resolving host names into IP addresses;
- a component for downloading documents using the HTTP protocol;
- a component for extracting links from HTML documents; and
- a component for determining whether a URL has been encountered before.

The remainder of this section describes how Mercator refines this basic algorithm.

Figure 1 shows Mercator's main components. Crawling is performed by multiple worker threads, typically numbering in the hundreds. Each worker repeatedly performs the steps needed to download and process a document. The first step of this loop ❶ is to remove an absolute URL from the shared URL frontier for downloading.

An absolute URL begins with a *scheme* (e.g., "http"), which identifies the network protocol that should be used to download it. In Mercator, these network protocols are implemented by *protocol modules*. The protocol modules to be used in a crawl are specified in a user-supplied configuration file, and are dynamically loaded at the start of the crawl. The default configuration includes protocol modules for HTTP, FTP, and Gopher.

Based on the URL's scheme, the worker selects the appropriate protocol module for downloading the document. It then invokes the protocol module's *fetch* method, which downloads the document from the Internet ❷ into a per-thread *RewindInputStream* ❸ (or RIS for short). A RIS is an I/O abstraction that is initialized from an arbitrary input stream, and that subsequently allows that stream's contents to be re-read multiple times.

Courteous web crawlers implement the Robots Exclusion Protocol, which allows web masters to declare parts of their sites off limits to crawlers [17]. The Robots Exclusion Protocol requires a web crawler to fetch a resource named "/robots.txt" containing these declarations from a web site before downloading any real content from it. To avoid downloading this resource on every request, Mercator's HTTP protocol module maintains a fixed-sized cache mapping host names to their robots exclusion rules. By default, the cache is limited to 2^{18} entries, and uses an LRU replacement strategy.

Once the document has been written to the RIS, the worker thread invokes the *content-seen test* to determine whether this document with the same content, but a different URL, has been seen before ❹. If so, the document is not processed any further, and the worker thread goes back to step ❶.

Every downloaded document has a *content type*. In addition to associating schemes with protocol modules, a Mercator configuration file also associates content types with one or more *processing modules*. A processing module is an abstraction for processing downloaded documents, for instance extracting links from HTML pages, counting the tags found in HTML pages, or collecting statistics about GIF images. In general, processing modules may have side-effects on the state of the crawler, as well as on their own internal state.

Based on the downloaded document's content type, the worker invokes the *process* method of each processing module associated with that content type ❺. For example, the Link Extractor and Tag Counter processing modules in Figure 1 are used for text/html documents, and the GIF Stats module is used for image/gif documents.

By default, a processing module for extracting links is associated with the content type text/html. The *process* method of this module extracts all links from an HTML page. Each link is converted into an absolute URL and tested against a user-supplied *URL filter* to determine if it should be downloaded ❻. If the URL passes the filter, it is submitted to the *duplicate URL eliminator* (DUE) ❼, which checks if the URL has been seen before, namely, if it is in the URL frontier or has already been downloaded. If the URL is new, it is added to the frontier ❽.

Finally, in the case of continuous crawling, the URL of the document that was just downloaded is also added back to the URL frontier. As noted earlier, a mechanism is required in the continuous crawling case for interleaving the downloading of new and old URLs. Mercator uses a randomized priority-based scheme for this purpose. A standard configuration for continuous crawling typically uses a frontier implementation that attaches priorities to URLs based on their download history, and whose dequeue method is biased towards higher priority URLs. Both the degree of bias and the algorithm for computing URL priorities are pluggable components. In one of our configurations, the priority of documents that do not change from one download to the next decreases over time, thereby causing them to be downloaded less frequently than documents that change often.

In addition to the numerous worker threads that download and process documents, every Mercator crawl also has a single background thread that performs a variety of tasks. The background thread wakes up periodically (by default, every 10 seconds), logs summary statistics about the crawl's progress, checks if the crawl should be terminated (either because the frontier is empty or because a user-specified time limit has been exceeded), and checks to see if it is time to checkpoint the crawl's state to stable storage.

Checkpointing is an important part of any long-running process such as a web crawl. By *checkpointing* we mean writing a representation of the crawler's state to stable storage that, in the event of a failure, is sufficient to allow the crawler to recover its state by reading the checkpoint and to resume crawling from the exact state it was in at the time of the checkpoint. By this definition, in the event of a failure, any work performed after the most recent checkpoint is lost, but none of the work up to the most recent checkpoint. In Mercator, the frequency with which the background thread performs a checkpoint is user-configurable; we typically checkpoint anywhere from 1 to 4 times per day.

The description so far assumed the case in which all Mercator threads are run in a single process. However, Mercator can be configured as a multi-process distributed system. In this configuration, one process is designated the *queen*, and the others are *drones*.[1] Both the queen and the drones run worker threads, but only the queen runs a background thread responsible for logging statistics, terminating the crawl, and initiating checkpoints.

In its distributed configuration, the space of host names is partitioned among the queen and drone processes. Each process is responsible only for the subset

[1]This terminology was inspired by the common practice of referring to web crawlers as spiders. In fact, our internal name for the distributed version of Mercator is Atrax, after *atrax robustus*, also known as the Sydney Funnel Web Spider, one of the few spider species that lives in colonies.

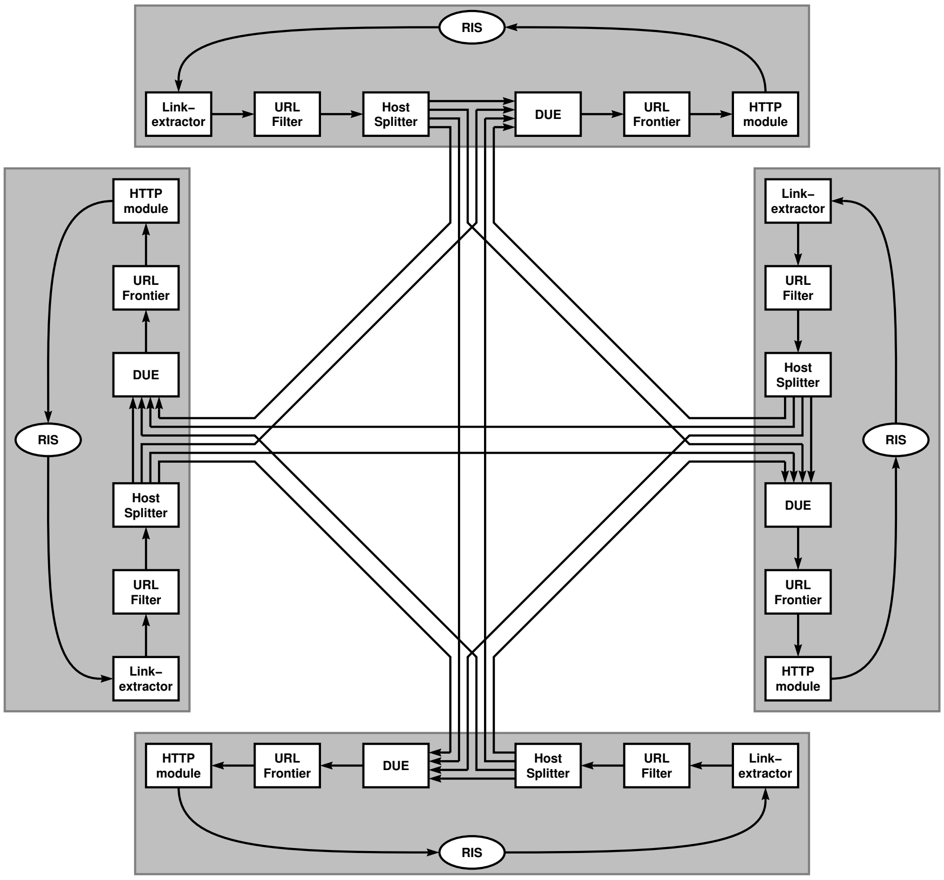


Figure 2: A four-node distributed crawling hive

of host names assigned to it. Hence, the central data structures of each crawling process — the URL frontier, the URL set maintained by the DUE, the DNS cache, etc. — contain data only for its hosts. Put differently, the state of a Mercator crawl is fully partitioned across the queen and drone processes; there is no replication of data.

In a distributed crawl, when a Link Extractor extracts a URL from a downloaded page, that URL is passed through the URL Filter, into a *host splitter* component. This component checks if the URL's host name is assigned to this process or not. Those that are assigned to this process are passed on to the DUE; the others are routed to the appropriate peer process, where it is then passed to that process's DUE component. Since about 80% of links are relative, the vast majority of discovered URLs remain local to the crawling process that discovered them. Moreover, Mercator buffers the outbound URLs so that they may be transmitted in batches for efficiency. Figure 2 illustrates this design.

The above description omits several important implementation details. Designing data structures that can efficiently handle hundreds of millions of entries poses many engineering challenges. Central to these concerns are the need to balance memory use and performance. The following subsections provide additional details about the URL frontier, the DUE, and the DNS resolver components. A more detailed description of the architecture and implementation is available elsewhere [12, 19].

3.1 A Polite and Prioritizing URL Frontier

Every web crawler must keep track of the URLs to be downloaded. We call the data structure for storing these URLs the *URL frontier*. Despite the name, Mercator's URL frontier actually stores objects that encapsulate both a URL and the download history of the corresponding document.

Abstractly speaking, a frontier is a URL repository that provides two major methods to its clients: one for adding a URL to the repository, and one for obtaining a URL from it. Note that the clients control in what order URLs are added, while the frontier controls in what order they are handed back out. In other words, the URL frontier controls the crawler's download schedule.

Like so many other parts of Mercator, the URL frontier is a pluggable component. We have implemented about half a dozen versions of this component. The main difference between the different versions lies in their scheduling policies. The policies differ both in complexity and in the degree of "politeness" (i.e., rate-limiting) they provide to the crawled web servers.

Most crawlers work by performing a breadth-first traversal of the web, starting from the pages in the seed set. Such traversals are easily implemented by using a first-in/first-out (FIFO) queue. However, the prevalence of relative URLs on web pages causes a high degree of host locality within the FIFO queue; that is, the queue contains runs of URLs with the same host name. If all of the crawler's threads dequeue URLs from a single FIFO queue, many of them will issue HTTP requests to the same web server simultaneously, thereby overloading it. Such behavior is considered socially unacceptable (in fact, it has the potential to crash some web servers).

Such overloads can be avoided by limiting the number of outstanding HTTP requests to any given web server. One way to achieve this is by ensuring that at any given time, only one thread is allowed to contact a particular web server. We call this the *weak politeness guarantee*.

We implemented a frontier that met the weak politeness guarantee and used it to perform several crawls, each of which fetched tens of millions of documents. During each crawl, we received a handful of complaints from various web server administrators. It became clear that our weak politeness guarantee was still considered too rude by some. The problem is that the weak politeness guarantee does not prevent a stream of requests from being issued to the same host without any pauses between them.

Figure 3 shows our most sophisticated frontier implementation. In addition to providing a stronger politeness guarantee that rate-limits the stream of HTTP requests issued to any given host, it also distributes the work among the crawling threads as evenly as possible (subject to the politeness requirement), and it provides a priority-based scheme for scheduling URL downloads. The frontier consists of a front-end (the top part of the figure) that is responsible for prioritizing URLs, and a back-end (the bottom part of the figure) that is responsible for ensuring strong politeness.

When a URL u is added to the frontier, a pluggable *prioritizer* component computes a priority value p between 1 and k based on the URL and its download history (e.g. whether the document has changed since the last download), and inserts u into front-end FIFO queue p.

The back-end maintains n FIFO queues, each of which is guaranteed to be non-empty and to contain URLs of only a single host, and a table T that maintains a map from hosts to back-end queues. Moreover, it maintains a heap data structure that contains a handle to each FIFO queue, and that is indexed by a timestamp indicating when the web server corresponding to the queue may be contacted again. Obtaining a URL from the frontier involves the following steps: First, the calling thread removes the root item from the heap (blocking, if necessary, until its timestamp is in the past). It then returns the head URL u from the corresponding back-end queue q. The calling thread will subsequently download the corresponding document.

Once the download has completed, u is removed from q. If q becomes empty, the calling thread refills q from the front end. This is done by choosing a front-end queue at random with a bias towards "high-priority" queues, and removing a URL u' from it. If one of the other back-end queues contains URLs with the same host component as u', u' is inserted into that queue and process of refilling q goes on. Otherwise, u' is added to q, and T is updated accordingly. Also, the calling thread computes the time at which u's host may be contacted again, and reinserts the handle to q with that timestamp back into the heap.

Note that the number n of back-end queues and the degree of rate-limiting go hand in hand: The larger the degree of rate-limiting, the more back-end queues are required to keep all the crawling threads busy. In our production crawls, we typically use 3 times as many back-end queues as crawling threads, and we wait 10

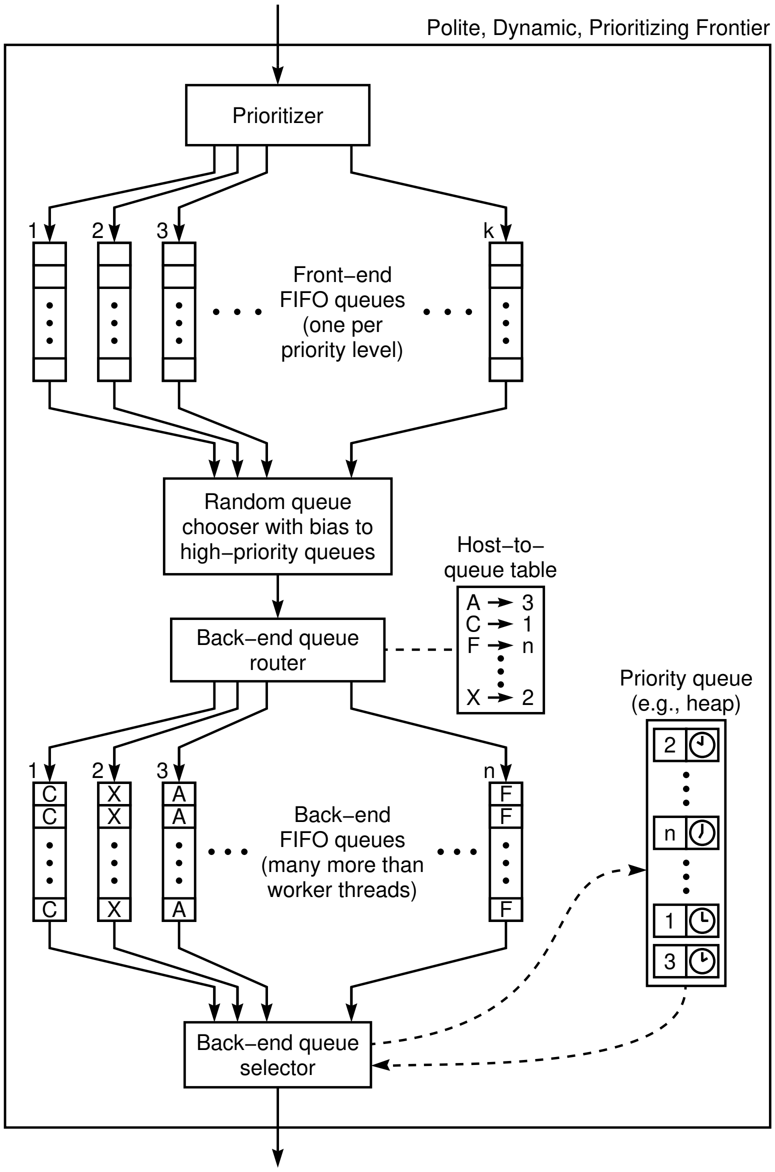


Figure 3: Our best URL frontier implementation

times as long as it took us to download a URL from a host before contacting that host again. These values are sufficient to keep all threads busy and to keep the rate of complaints to a bare minimum.

In a crawl of the entire web, the URL frontier soon outgrows the available memory of even the largest machines. It is therefore necessary to store most of the frontier's URLs on disk. In Mercator, each of the FIFO queues stores the bulk of its URLs on disk, and buffers only a fixed number of the URLs at its head and tail in memory.

3.2 Efficient Duplicate URL Eliminators

In the course of extracting links, any web crawler will encounter multiple links to the same document. To avoid downloading and processing a document multiple times, a duplicate URL eliminator (DUE) guards the URL frontier. Extracted links are *submitted* to the DUE, which passes new ones to the frontier while ignoring those that have been submitted to it before.

One of our implementations of the DUE maintains an in-memory hash table of all URLs that have been encountered before. To save space, the table stores 8-byte checksums of the URLs rather than the URLs themselves. We compute the checksums using Rabin's fingerprinting algorithm [3, 21], which has good spectral properties and gives exponentially small probabilistic bounds on the likelihood of a collision. The high-order 3 bytes of the checksum are used to index into the hash table spine. Since all checksums in the same overflow bucket would have the same high-order 3 bytes, we actually store only the 5 low-order bytes. Taking pointer and counter overhead into account, storing the checksums of 1 billion URLs in such a hash table requires slightly over 5 GB.

This implementation is very efficient, but it requires a substantial hardware investment; moreover, the memory requirements are proportional to the size of the crawl. A disk-based approach avoids these problems, but is difficult to implement efficiently. Our first disk-based implementation essentially stored the URL fingerprint hash table on disk. By caching popular fingerprints in memory, only one in every six DUE submissions required a disk access. However, due to the spectral properties of the fingerprinting function, there is very little locality in the stream of fingerprints that miss on the in-memory cache, so virtually every disk access required a disk seek. On state-of-the-art disks, the average seek requires about 8 ms, which would enable us to perform 125 seeks or 750 DUE submissions per second. Since the average web page contains about 10 links, this would limit the crawling rate to 75 downloads per second. Initially, this bottleneck is masked by the operating system's file buffer cache, but once the disk-based hash table grows larger than the file buffer cache, the seek operations become the performance bottleneck.

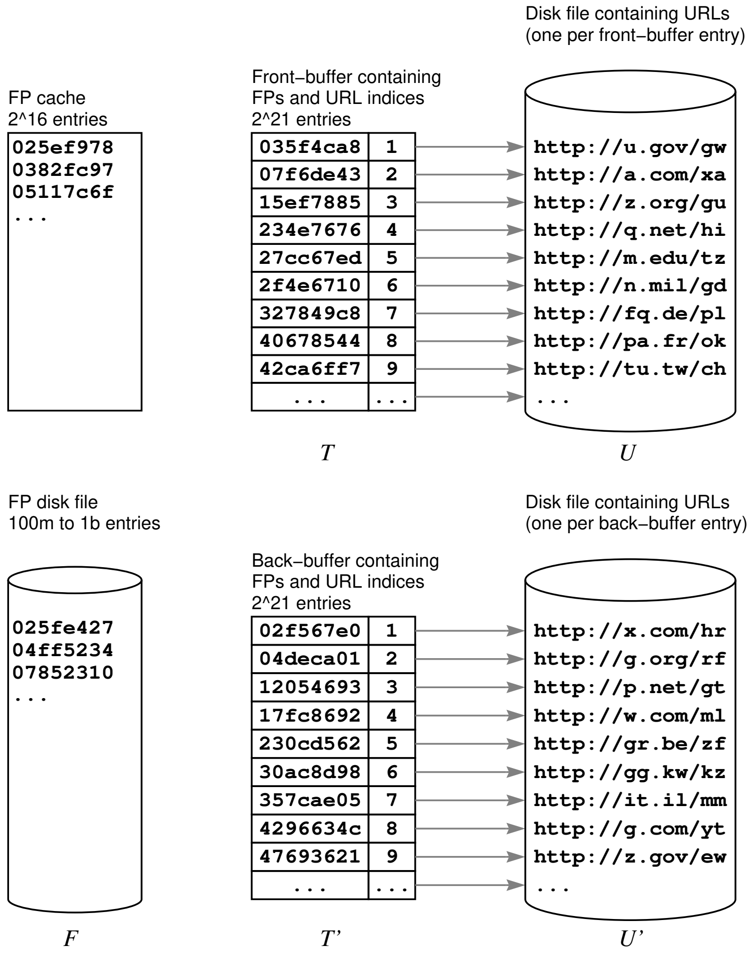


Figure 4: Our most efficient disk-based DUE implementation

Both designs described so far add never-before-seen URLs to the frontier *immediately*. By buffering URLs, we can amortize the access to the disk, thereby increasing the throughput of the DUE. Figure 4 shows the main data structures of our most efficient disk-based DUE implementation. This implementation's largest data structure is a file F of sorted URL fingerprints.

When a URL u is submitted to the DUE, its fingerprint fp is computed. Next, fp is checked against a cache of popular URLs and an in-memory hash table T. If fp is contained in either, no further action is required. Otherwise, u is appended to a URL disk file U, and a mapping from fp to u's ordinal in U is added to T.

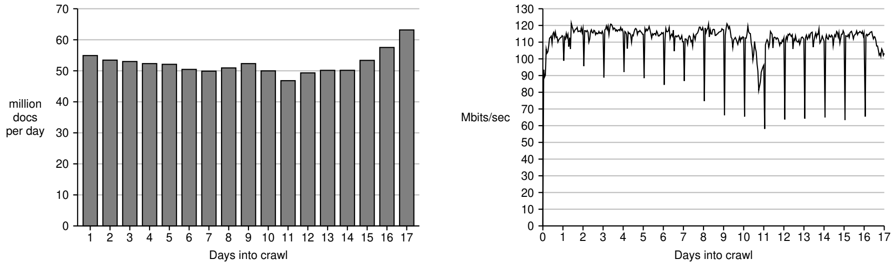
Once T's size exceeds a predefined limit, the thread that "broke the camel's back" atomically copies T's content to a table T' consisting of fingerprint/ordinal pairs, empties T, and renames U to U'. Once this very short atomic operation completes, other crawling threads are free to submit URLs to the DUE, while the back-breaking thread adds the new content of T' and U' into F and the frontier, respectively. It first sorts T' by fingerprint value, and then performs a linear merge of T' and F, *marking* every row in T' whose fingerprint was added to F. Next, it sorts T' by ordinal value. Finally, it scans both T' and U' sequentially, and adds all URLs in U' that are marked in T' to the frontier.

In real crawls, we found that this DUE implementation performs at least twice as well as the one that is seek-limited; however, its throughput still deteriorates over time, because the time needed to merge T' into F eventually exceeds the time required to fill T.

3.3 The Trouble with DNS

Hosts on the Internet are identified by Internet Protocol (IP) addresses, which are 32-bit numbers. IP addresses are not mnemonic. This problem is avoided by the use of symbolic host names, such as *cnn.com*, which identify one or more IP addresses. Any program that contacts sites on the internet whose identities are provided in the form of symbolic host names must resolve those names into IP addresses. This process is known as *host name resolution*, and it is supported by the *domain name service* (DNS). DNS is a globally distributed service in which name servers refer requests to more authoritative name servers until an answer is found. Therefore, a single DNS request may take seconds or even tens of seconds to complete, since it may require many round-trips across the globe.

DNS name resolution is a well-documented bottleneck of most web crawlers. We tried to alleviate this bottleneck by caching DNS results, but that was only partially effective. After some probing, we discovered that the Java interface to DNS lookups is synchronized. Further investigation revealed that the DNS interface on most flavors of Unix (i.e., the `gethostbyname` function provided as part



(a) Documents downloaded per day (b) Megabits downloaded per second

Figure 5: Mercator's performance over a 17-day crawl

of the Berkeley Internet Name Domain (BIND) distribution [1]) is also synchronized. This meant that only one DNS request per address space on an uncached name could be outstanding at once. The cache miss rate is high enough that this limitation causes a severe bottleneck.

To work around these problems, we made DNS resolution one of Mercator's pluggable components. We implemented a multi-threaded DNS resolver component that does not use the resolver provided by the host operating system, but rather directly forwards DNS requests to a local name server, which does the actual work of contacting the authoritative server for each query. Because multiple requests can be made in parallel, our resolver can resolve host names much more rapidly than either the Java or Unix resolvers.

This change led to a significant crawling speedup. Before making the change, performing DNS lookups accounted for 70% of each thread's elapsed time. Using our custom resolver reduced that elapsed time to 14%. (Note that the actual number of CPU cycles spent on DNS resolution is extremely low. Most of the elapsed time is spent waiting for remote DNS servers.) Moreover, because our resolver can perform resolutions in parallel, DNS is no longer a bottleneck; if it were, we would simply increase the number of worker threads.

4 Experiences from a Large Crawl

This section describes the results of our crawling experiments. Our crawling cluster consists of four Compaq DS20E AlphaServers, each one equipped with 4 GB of main memory, 650 GB of disk, and a 100 Mbit/sec Ethernet card. The cluster is located close to the Internet backbone. Our ISP rate-limits our bandwidth to 160 Mbits/sec.

In December 2000, we performed a crawl that processed 891 million URLs

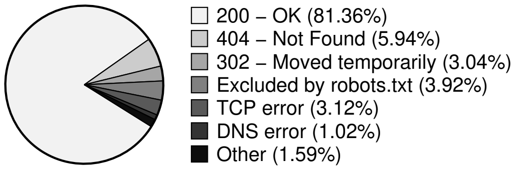


Figure 6: Outcome of download attempts

over the course of 17 days.[2] Figure 5a shows the number of URLs processed per day of the crawl; Figure 5b shows the bandwidth consumption over the life of the crawl. The periodic downspikes are caused by the crawler checkpointing its state once a day. The crawl was network-limited over its entire life; CPU load was below 50%, and disk activity was low as well.

As any web user knows, not all download attempts are successful. During our crawl, we collected statistics about the outcome of each download attempt. Figure 6 shows the outcome percentages. Of the 891 million processed URLs, 35 million were excluded from download by robots.txt files, and 9 million referred to a nonexistent web server; in other words, the crawler performed 847 million HTTP requests. 725 million of these requests returned an HTTP status code of 200 (i.e., were successful), 94 million returned an HTTP status code other than 200, and 28 million encountered a TCP failure.

There are many different types of content on the internet, such as HTML pages, GIF and JPEG images, MP3 audio files, and PDF documents. The MIME (Multipurpose Internet Mail Extensions) standard defines a naming scheme for these content types [8]. We have collected statistics about the distribution of content types of the successfully downloaded documents. Overall, our crawl discovered 3,173 different content types (many of which are misspellings of common content types). Figure 7 shows the percentages of the the most common types. HTML

[2] As a point of comparison, the current Google index contains about 700 million fully-indexed pages (the index size claimed on the Google home page – 1.35 billion — includes URLs that have been discovered, but not yet downloaded).

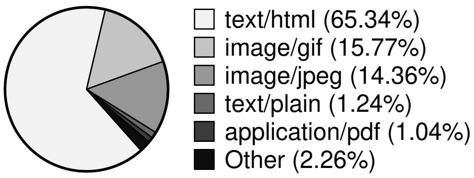


Figure 7: Distribution of content types

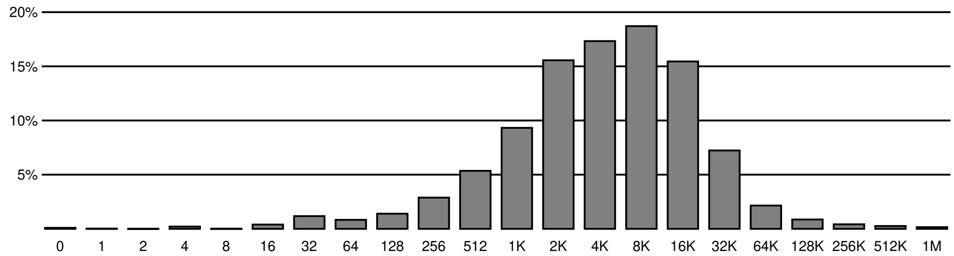


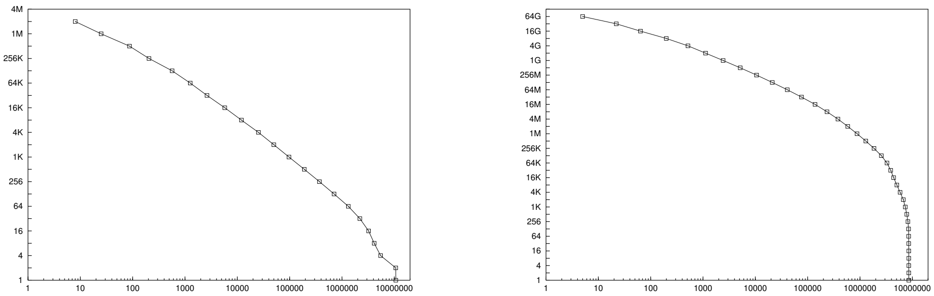
Figure 8: Distribution of document sizes

pages (of type *text/html*) account for nearly two-thirds of all documents; images (in both GIF and JPEG formats) account for another 30%; all other content types combined account for less than 5%.

Figure 8 is a histogram showing the document size distribution. In this figure, the documents are distributed over 22 bins labeled with exponentially increasing document sizes; a document of size n is placed in the rightmost bin with a label not greater than n. Of the 725 million documents that were successfully downloaded, 67% were between 2K and 32K bytes in size, corresponding to the four tallest bars in the figure.

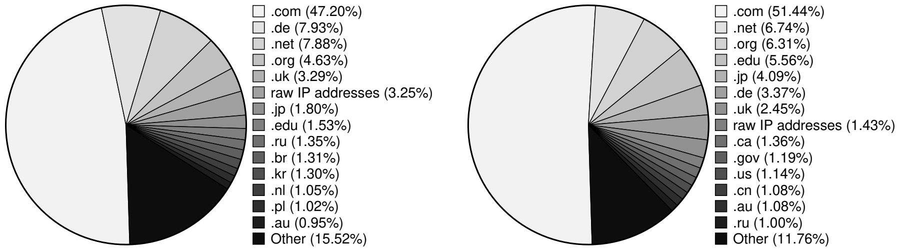
Figure 9 shows the distribution of content across web servers. Figure 9a measures the content using a granularity of whole pages, while Figure 9b measures content in bytes. Both figures are plotted on a log-log scale, and in both, a point (x, y) indicates that x web servers had at least y pages/bytes. The near-linear shape of the plot in Figure 9a indicates that the distribution of pages over web servers is Zipfian.

Finally, Figure 10 shows the distributions of web servers and web pages across top-level domains. About half of the servers and pages fall into the .com domain. For the most part, the numbers of hosts and pages in a top-level domain are well-



(a) Distribution of pages over web servers (b) Distribution of bytes over web servers

Figure 9: Document and web server size distributions



(a) Distribution of hosts over top-level domains

(b) Distribution of pages over top-level domains

Figure 10: Distribution of hosts and pages over top-level domains

correlated. However, there are some interesting wrinkles. For example, the .edu domain contains only about 1.53% of the hosts, but 5.56% of the total pages. In other words, the average university web server contains almost four times as many pages as the average server on the web at large.

5 Conclusion

High-performance web crawlers are an important component of many web services. Building a high-performance crawler is a non-trivial endeavor: the data manipulated by the crawler is too big to fit entirely in memory, so there are performance issues related to how to balance the use of disk and memory. This chapter has enumerated the main components required in any crawler, and it has discussed design alternatives for some of those components. In particular, the chapter described Mercator, an extensible, distributed, high-performance crawler written entirely in Java.

Mercator's design features a crawler core for handling the main crawling tasks, and extensibility through a component-based architecture that allows users to supply new modules at run-time for performing customized crawling tasks. These extensibility features have been quite successful. We were able to adapt Mercator to a variety of crawling tasks, and the new code was typically quite small (tens to hundreds of lines). Moreover, the flexibility afforded by the component model encouraged us to experiment with different implementations of the same functional components, and thus enabled us to discover new and efficient data structures. In our experience, these innovations produced larger performance gains than low-level tuning of our user-space code [13].

Mercator's scalability design has also worked well. It is easy to configure the crawler for varying memory footprints. For example, we have run it on machines with memory sizes ranging from 128 MB to 2 GB. The ability to configure Mercator for a wide variety of hardware platforms makes it possible to select the most cost-effective platform for any given crawling task.

Although our use of Java as an implementation language was met with considerable scepticism when we began the project, we have not regretted the choice. Java's combination of features — including threads, garbage collection, objects, and exceptions — made our implementation easier and more elegant. Moreover, on I/O-intensive applications, Java has little negative impact on performance. Profiling Mercator running on Compaq AlphaServers reveals that over 60% of the cycles are spent in the kernel and in C libraries; less than 40% are spent executing (JIT-compiled) Java bytecode. In fact, Mercator is faster than any other web crawler for which performance numbers have been published.

Mercator has proven to be extremely popular. It has been incorporated into AltaVista's Search Engine 3 product, and it is being used as the web crawler for AltaVista's American and European search sites. Our colleague Raymie Stata has performed Mercator crawls that collected over 12 terabytes of web content, which he has contributed to the Internet Archive's web page collection. Raymie also performed a continuous crawl on behalf of the Library of Congress to monitor coverage of the 2000 U.S. Presidential election. That crawl took daily snapshots of about 200 election-related web sites during the five months preceding the inauguration. Finally, Mercator has been an enabler for other web research within Compaq. For example, we have configured Mercator to perform random walks instead of breadth-first search crawls, using the crawl traces to estimate the quality and sizes of major search engine indices [10, 11].

References

[1] Berkeley Internet Name Domain (BIND).
http://www.isc.org/bind.html

[2] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, pages 107–117, April 1998.

[3] Andrei Broder. Some Applications of Rabin's Fingerprinting Method. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[4] Mike Burner. Crawling towards Eternity: Building an archive of the World Wide Web. Web Techniques Magazine, 2(5), May 1997.

[5] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through URL ordering. In *Proceedings of the Seventh International World Wide Web Conference*, pages 161–172, April 1998.

[6] Jenny Edwards, Kevin McCurley, and John Tomlin. An Adaptive Model for Optimizing Performance of an Incremental Web Crawler. In *Proceedings of the Tenth International World Wide Web Conference*, pages 106–113, May 2001.

[7] David Eichmann. The RBSE Spider – Balancing Effective Search Against Web Load. In *Proceedings of the First International World Wide Web Conference*, pages 113–120, 1994.

[8] Ned Freed and Nathaniel Borenstein. Multipurpose Internet Mail Extensions Part Two: Media Types. RFC 2046, Network Working Group, Internet Engineering Task Force (IETF), November 1996.

[9] Matthew Gray. Internet Growth and Statistics: Credits and Background. http://www.mit.edu/people/mkgray/net/background.html

[10] Monika Henzinger, Allan Heydon, Michael Mitzenmacher, and Marc A. Najork. Measuring Index Quality using Random Walks on the Web. In *Proceedings of the Eighth International World Wide Web Conference*, pages 213–225, May 1999.

[11] Monika Henzinger, Allan Heydon, Michael Mitzenmacher, and Marc A. Najork. On Near-Uniform URL Sampling. In *Proceedings of the Ninth International World Wide Web Conference*, pages 295–308, May 2000.

[12] Allan Heydon and Marc Najork. Mercator: A scalable, extensible Web crawler. *World Wide Web*, **2**, pages 219–229, December 1999.

[13] Allan Heydon and Marc Najork. Performance Limitations of the Java Core Libraries. *Concurrency: Practice and Experience*, **12**:363-373, May 2000.

[14] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. WebBase: A repository of Web pages. In *Proceedings of the Ninth International World Wide Web Conference*, pages 277–293, May 2000.

[15] The Internet Archive.
`http://www.archive.org/`

[16] Martijn Koster. The Web Robots Pages.
`http://info.webcrawler.com/mak/projects/robots/robots.html`

[17] Martijn Koster. A Method for Web Robots Control. Network Working Group, Internet Draft, December 1996.
`http://www.robotstxt.org/wc/norobots-rfc.html`

[18] Oliver A. McBryan. GENVL and WWWW: Tools for Taming the Web. In *Proceedings of the First International World Wide Web Conference*, pages 79–90, 1994.

[19] Marc Najork and Allan Heydon. On High-Performance Web Crawling. *SRC Research Report*, Compaq Systems Research Center, forthcoming.

[20] Brian Pinkerton. Finding What People Want: Experiences with the WebCrawler. In *Proceedings of the Second International World Wide Web Conference*, 1994.

[21] Michael O. Rabin. Fingerprinting by Random Polynomials. Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.