Design and Implementation of a High-Performance Distributed Web Crawler*

Vladislav Shkapenyuk ***Torsten Suel***

CIS Department
Polytechnic University
Brooklyn, NY 11201

vshkap@research.att.com, suel@poly.edu

Abstract

Broad web search engines as well as many more specialized search tools rely on web crawlers to acquire large collections of pages for indexing and analysis. Such a web crawler may interact with millions of hosts over a period of weeks or months, and thus issues of robustness, flexibility, and manageability are of major importance. In addition, I/O performance, network resources, and OS limits must be taken into account in order to achieve high performance at a reasonable cost.

In this paper, we describe the design and implementation of a distributed web crawler that runs on a network of workstations. The crawler scales to (at least) several hundred pages per second, is resilient against system crashes and other events, and can be adapted to various crawling applications. We present the software architecture of the system, discuss the performance bottlenecks, and describe efficient techniques for achieving high performance. We also report preliminary experimental results based on a crawl of 120 *million pages on* 5 *million hosts.*

1 Introduction

The World Wide Web has grown from a few thousand pages in 1993 to more than two billion pages at present. Due to this explosion in size, web search engines are becoming increasingly important as the primary means of locating relevant information. Such search engines rely on massive collections of web pages that are acquired with the help of *web crawlers*, which traverse the web by following hyperlinks and storing downloaded pages in a large database that is later indexed for efficient execution of user queries. Many researchers have looked at web search technology over the last few years, including crawling strategies, storage, indexing, ranking techniques, and a significant amount of work on the structural analysis of the web and web graph; see [1, 7] for overviews of some recent work and [26, 2] for basic techniques.

Thus, highly efficient crawling systems are needed in order to download the hundreds of millions of web pages indexed by the major search engines. In fact, search engines compete against each other primarily based on the size and currency of their underlying database, in addition to the quality and response time of their ranking function. Even the largest search engines, such as Google or AltaVista, currently cover only limited parts of the web, and much of their data is several months out of date. (We note, however, that crawling speed is not the only obstacle to increased search engine size, and that the scaling of query throughput and response time to larger collections is also a major issue.)

A crawler for a large search engine has to address two issues. First, it has to have a good crawling strategy, i.e., a strategy for deciding which pages to download next. Second, it needs to have a highly optimized system architecture that can download a large number of pages per second while being robust against crashes, manageable, and considerate of resources and web servers. There has been some recent academic interest in the first issue, including work on strategies for crawling important pages first [12, 21], crawling pages on a particular topic or of a particular type [9, 8, 23, 13], recrawling (refreshing) pages in order to optimize the overall "freshness" of a collection of pages [11, 10], or scheduling of crawling activity over time [25].

In contrast, there has been less work on the second issue. Clearly, all the major search engines have highly optimized crawling systems, although details of these systems are usually proprietary. The only system described in detail in the literature appears to be the *Mercator* system of Heydon and Najork at DEC/Compaq [16], which is used by AltaVista. (Some details are also known about the first version of the Google crawler [5] and the system used by the Internet Archive [6].) While it is fairly easy to build a slow crawler that downloads a few pages per second for a short period of time, building a high-performance system that can download hundreds of millions of pages over several weeks presents a number of challenges in system design, I/O and

*Work supported by NSF CAREER Award NSF CCR-0093400, Intel Corporation, and the New York State Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University, and by equipment grants from Intel Corporation and Sun Microsystems.

network efficiency, and robustness and manageability.

Most of the recent work on crawling strategies does not address these performance issues at all, but instead attempts to minimize the number of pages that need to be downloaded, or maximize the benefit obtained per downloaded page. (An exception is the work in [8] that considers the system performance of a focused crawler built on top of a general-purpose database system, although the throughput of that system is still significantly below that of a high-performance bulk crawler.) In the case of applications that have only very limited bandwidth that is acceptable. However, in the case of a larger search engine, we need to combine good crawling strategy and optimized system design.

In this paper, we describe the design and implementation of such an optimized system on a network of workstations. The choice of crawling strategy is largely orthogonal to our work. We describe the system using the example of a simple breadth-first crawl, although the system can be adapted to other strategies. We are primarily interested in the I/O and network efficiency aspects of such a system, and in scalability issues in terms of crawling speed and number of participating nodes. We are currently using the crawler to acquire large data sets for work on other aspects of web search technology such as indexing, query processing, and link analysis. We note that high-performance crawlers are currently not widely used by academic researchers, and hence few groups have run experiments on a scale similar to that of the major commercial search engines (one exception being the WebBase project [17] and related work at Stanford). There are many interesting questions in this realm of massive data sets that deserve more attention by academic researchers.

1.1 Crawling Applications

There are a number of different scenarios in which crawlers are used for data acquisition. We now describe a few examples and how they differ in the crawling strategies used.

Breadth-First Crawler: In order to build a major search engine or a large repository such as the Internet Archive [18], high-performance crawlers start out at a small set of pages and then explore other pages by following links in a "breadth first-like" fashion. In reality, the web pages are often not traversed in a strict breadth-first fashion, but using a variety of policies, e.g., for pruning crawls inside a web site, or for crawling more important pages first[1].

Recrawling Pages for Updates: After pages are initially acquired, they may have to be periodically recrawled and checked for updates. In the simplest case, this could be done by starting another broad breadth-first crawl, or by simply requesting all URLs in the collection again. However, a variety of heuristics can be employed to recrawl more important pages, sites, or domains more frequently. Good recrawling strategies are crucial for maintaining an up-to-date search index with limited crawling bandwidth, and recent work by Cho and Garcia-Molina [11, 10] has studied techniques for optimizing the "freshness" of such collections given observations about a page's update history.

Focused Crawling: More specialized search engines may use crawling policies that attempt to focus only on certain types of pages, e.g., pages on a particular topic or in a particular language, images, mp3 files, or computer science research papers. In addition to heuristics, more general approaches have been proposed based on link structure analysis [9, 8] and machine learning techniques [13, 23]. The goal of a focused crawler is to find many pages of interest without using a lot of bandwidth. Thus, most of the previous work does not use a high-performance crawler, although doing so could support large specialized collections that are significantly more up-to-date than a broad search engine.

Random Walking and Sampling: Several techniques have been studied that use random walks on the web graph (or a slightly modified graph) to sample pages or estimate the size and quality of search engines [3, 15, 14].

Crawling the "Hidden Web": A lot of the data accessible via the web actually resides in databases and can only be retrieved by posting appropriate queries and/or filling out forms on web pages. Recently, a lot of interest has focused on automatic access to this data, also called the "Hidden Web", "Deep Web", or "Federated Facts and Figures". Work in [22] has looked at techniques for crawling this data. A crawler such as the one described here could be extended and used as an efficient front-end for such a system. We note, however, that there are many other challenges associated with access to the hidden web, and the efficiency of the front end is probably not the most important issue.

1.2 Basic Crawler Structure

Given these scenarios, we would like to design a flexible system that can be adapted to different applications and strategies with a reasonable amount of work. Note that there are significant differences between the scenarios. For example, a broad breadth-first crawler has to keep track of which pages have been crawled already; this is commonly done using a *"URL seen"* data structure that may have to reside on disk for large crawls. A link analysis-based focused crawler, on the other hand, may use an additional data structure to represent the graph structure of the crawled part of the web, and a classifier to judge the relevance of a page [9, 8], but the size of the structures may be much smaller. On the other hand, there are a number of common tasks that need to be done in all or most scenarios, such as enforcement of robot exclusion, crawl speed control, or DNS resolution.

[1] See [12] for heuristics that attempt to crawl important pages first, and [21] for experimental results showing that even strict breadth-first will quickly find most pages with high Pagerank.

For simplicity, we separate our crawler design into two main components, referred to as *crawling application* and *crawling system*. The crawling application decides what page to request next given the current state and the previously crawled pages, and issues a stream of requests (URLs) to the crawling system. The crawling system (eventually) downloads the requested pages and supplies them to the crawling application for analysis and storage. The crawling system is in charge of tasks such as robot exclusion, speed control, and DNS resolution that are common to most scenarios, while the application implements crawling strategies such as “breadth-first” or ”focused“. Thus, to implement a focused crawler instead of a breadth-first crawler, we would use the same crawling system (with a few different parameter settings) but a significantly different application component, written using a library of functions for common tasks such as parsing, maintenance of the *“URL seen”* structure, and communication with crawling system and storage.

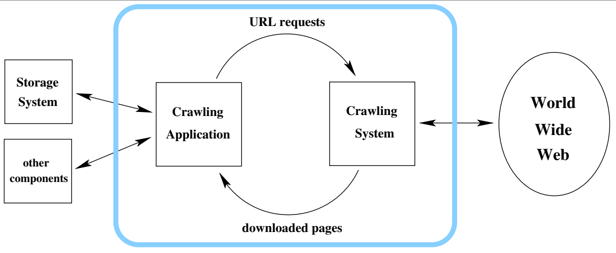


Figure 1: Basic two components of the crawler

At first glance, implementation of the crawling system may appear trivial. This is however not true in the high-performance case, where several hundred or even a thousand pages have to be downloaded per second. In fact, our crawling system consists itself of several components that can be replicated for higher performance. Both crawling system and application can also be replicated independently, and several different applications could issue requests to the same crawling system, showing another motivation for the design[2]. More details on the architecture are given in Section 2. We note here that this partition into application and system components is a design choice in our system, and not used by some other systems[3], and that it is not always obvious in which component a particular task should be handled. For the work in this paper, we focus on the case of a broad breadth-first crawler as our crawling application.

1.3 Requirements for a Crawler

We now discuss the requirements for a good crawler, and approaches for achieving them. Details on our solutions are given in the subsequent sections.

Flexibility: As mentioned, we would like to be able to use the system in a variety of scenarios, with as few modifications as possible.

Low Cost and High Performance: The system should scale to at least several hundred pages per second and hundreds of millions of pages per run, and should run on low-cost hardware. Note that efficient use of disk access is crucial to maintain a high speed after the main data structures, such as the *“URL seen”* structure and crawl frontier, become too large for main memory. This will only happen after downloading several million pages.

Robustness: There are several aspects here. First, since the system will interact with millions of servers, it has to tolerate bad HTML, strange server behavior and configurations, and many other odd issues. Our goal here is to err on the side of caution, and if necessary ignore pages and even entire servers with odd behavior, since in many applications we can only download a subset of the pages anyway. Secondly, since a crawl may take weeks or months, the system needs to be able to tolerate crashes and network interruptions without losing (too much of) the data. Thus, the state of the system needs to be kept on disk. We note that we do not really require strict ACID properties. Instead, we decided to periodically synchronize the main structures to disk, and to recrawl a limited number of pages after a crash.

Etiquette and Speed Control: It is extremely important to follow the standard conventions for robot exclusion (`robots.txt` and robots meta tags), to supply a contact URL for the crawler, and to supervise the crawl. In addition, we need to be able to control access speed in several different ways. We have to avoid putting too much load on a single server; we do this by contacting each site only once every 30 seconds unless specified otherwise. It is also desirable to throttle the speed on a domain level, in order not to overload small domains, and for other reasons to be explained later. Finally, since we are in a campus environment where our connection is shared with many other users, we also need to control the total download rate of our crawler. In particular, we crawl at low speed during the peek usage hours of the day, and at a much higher speed during the late night and early morning, limited mainly by the load tolerated by our main campus router.

Manageability and Reconfigurability: An appropriate interface is needed to monitor the crawl, including the speed of the crawler, statistics about hosts and pages, and the sizes of the main data sets. The administrator should be able to adjust the speed, add and remove components, shut down the system, force a checkpoint, or add hosts and domains to a “blacklist” of places that the crawler should avoid. After a crash or shutdown, the software of the system may be modified to fix problems, and we may want to continue the crawl

[2]Of course, in the case of the application, replication is up to the designer of the component, who has to decide how to partition data structures and workload.

[3]E.g., Mercator [16] does not use this partition, but achieves flexibility through the use of pluggable Java components.

using a different machine configuration. In fact, the software at the end of our first huge crawl was significantly different from that at the start, due to the need for numerous fixes and extensions that became only apparent after tens of millions of pages had been downloaded.

1.4 Content of this Paper

In this paper, we describe the design and implementation of a distributed web crawler that runs on a network of workstations. The crawler scales to (at least) several hundred pages per second, is resilient against system crashes and other events, and can be adapted to various crawling applications. We present the software architecture of the system, discuss the performance bottlenecks, and describe efficient techniques for achieving high performance. We also report preliminary experimental results based on a crawl of 120 million pages on 5 million hosts.

The remainder of this paper is organized as follows. Section 2 describes the architecture of our system and its major components, and Section 3 describes the data structures and algorithmic techniques that were used in more detail. Section 4 presents preliminary experimental results. Section 5 compares our design to that of other systems we know of. Finally, Section 6 offers some concluding remarks.

2 System Architecture

We now give a more detailed description of the architecture of our distributed crawler. As mentioned before, we partition the system into two major components - crawling system and crawling application. The crawling system itself consists of several specialized components, in particular a *crawl manager*, one or more *downloaders*, and one or more *DNS resolvers*. All of these components, plus the crawling application, can run on different machines (and operating systems) and can be replicated to increase the system performance. The crawl manager is responsible for receiving the URL input stream from the applications and forwarding it to the available downloaders and DNS resolvers while enforcing rules about robot exclusion and crawl speed. A downloader is a high-performance asynchronous HTTP client capable of downloading hundreds of web pages in parallel, while a DNS resolver is an optimized stub DNS resolver that forwards queries to local DNS servers.

An example of a small configuration with three downloaders is given in Figure 2, which also shows the main data flows through the system. This configuration is very similar to the one we used for our crawls, except that most of the time we used at most 2 downloaders. A configuration as the one in Figure 2 would require between 3 and 5 workstations, and would achieve an estimated peek rate of 250 to 400 HTML pages per second[4]. We discuss scaling in more detail further below.

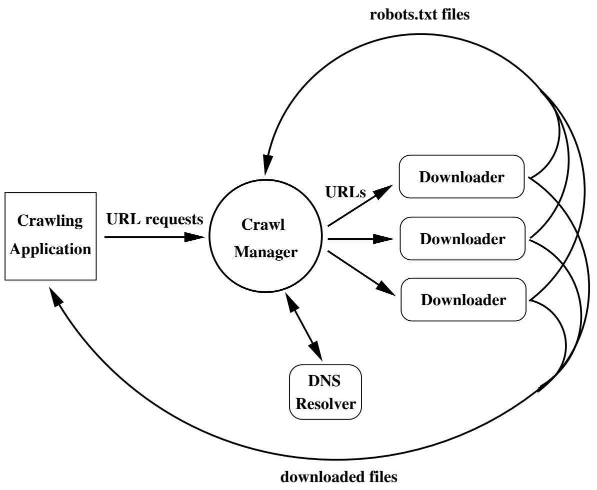


Figure 2: Small crawler configuration

Communication in our system is done in two ways: via sockets for small messages, and via file system (NFS) for larger data streams. The use of NFS in particular makes the design very flexible and allows us to tune system performance by redirecting and partitioning I/O between disks. While NFS has its performance limitations, we believe that the basic approach will scale well for networks of low-cost workstations and that if necessary it would be easy to switch to a more optimized file transfer mechanism. Not shown in Figure 2 are the interactions between crawling application and storage system, downloaders and web server, and between crawl manager and a separate web interface for system management. The entire system contains about 5000 lines of C++ and Python code. Implementation was started in July 2000 as part of the first author's Senior Project. The first significant crawls were performed in January 2001, and development is still continuing. We now give some more details on each components.

2.1 Crawl Manager

The crawl manager is the central component of the system, and the first component that is started up. Afterwards, other components are started and register with the manager to offer or request services. The manager is the only component visible to the other components, with the exception of the case of a parallelized application, described further below, where the different parts of the application have to interact. The manager receives requests for URLs from the application, where each request has a priority level, and a pointer to a file containing several hundred or thousand URLs and lo-

[4]The peek rate depends on the machine configuration and network connection. Due to limited network capacity, we are not able to run at more than 300 pages per second on our campus connection.

cated on some disk accessible via NFS. The manager will enqueue the request, and will eventually load the corresponding file in order to prepare for the download, though this is done as late as possible in order to limit the size of the internal data structures. In general, the goal of the manager is to download pages in approximately the order specified by the application, while reordering requests as needed to maintain high performance without putting too much load on any particular web server. This policy is formalized in Subsection 3.5.

After loading the URLs of a request files, the manager queries the DNS resolvers for the IP addresses of the servers, unless a recent address is already cached. The manager then requests the file `robots.txt` in the web server's root directory, unless it already has a recent copy of the file. This part was initially implemented as a separate component that acted as an application issuing requests for robot files with high priority back to the manager. It is now incorporated into the manager process, and the robot files are written to a separate directory from the other data so they can be accessed and parsed by the manager later (see Figure 2). Finally, after parsing the robots files and removing excluded URLs, the requested URLs are sent in batches to the downloaders, making sure that a certain interval between requests to the same server is observed. The manager later notifies the application of the pages that have been downloaded and are available for processing.

The manager is also in charge of limiting the overall speed of the crawl and balancing the load among downloaders and DNS resolvers, by monitoring and adjusting DNS resolver load and downloader speed as needed. The manager performs periodic snapshots of its data structures, and after a crash, a limited number of pages may have to be recrawled. It is up to the application to detect these duplicate pages. The crawl manager is implemented in C++ and uses Berkeley DB[5] and STL for the major data structures, which are described in more detail in Subsection 3.4.

2.2 Downloaders and DNS Resolvers

The downloader component, implemented in Python, fetches files from the web by opening up to 1000 connections to different servers, and polling these connections for arriving data. Data is then marshaled into files located in a directory determined by the application and accessible via NFS. Since a downloader often receives more than a hundred pages per second, a large number of pages have to be written out in one disk operation. We note that the way pages are assigned to these data files is unrelated to the structure of the request files sent by the application to the manager. Thus, it is up to the application to keep track of which of its URL requests have been completed. The manager can adjust the speed of a downloader by changing the number of concurrent connections that are used.

The DNS resolver, implemented in C++, is also fairly simple. It uses the GNU `adns` asynchronous DNS client library[6] to access a DNS server usually collocated on the same machine. While DNS resolution used to be a significant bottleneck in crawler design due to the synchronous nature of many DNS interfaces, we did not observe any significant performance impacts on our system while using the above library. However, DNS lookups generate a significant number of additional frames of network traffic, which may restrict crawling speeds due to limited router capacity.

2.3 Crawling Application

As mentioned, the crawling application we consider in this paper is a breadth-first crawl starting out at a set of seed URLs, in our case the URLs of the main pages of several hundred US Universities, which are initially sent to the crawl manager. The application then parses each downloaded page for hyperlinks, checks whether these URLs have already been encountered before, and if not, sends them to the manager in batches of a few hundred or thousand. The downloaded files are then forwarded to a storage manager for compression and storage in a repository. The crawling application is implemented in C++ using STL and the Red-Black tree implementation in the `kazlib` library[7]. (The application consists of two threads each using a Red-Black tree data structure; this required use of two different implementations since the current implementation in STL is not thread-safe.)

The data structure and performance aspects of the application will be discussed in detail in Subsection 3.2. We note however the following important two points: First, since each page contains on average about 8 hyperlinks, the set of encountered (but not necessarily downloaded) URLs will grow very quickly beyond the size of main memory, even after eliminating duplicates. Thus, after downloading 20 million pages, the size of the set of encountered URLs will be well above 100 million. Second, at this point, any hyperlink parsed from a newly downloaded page and sent to the manager will only be downloaded several days or weeks later. Thus, there is no reason for the manager to immediately insert new requests into its dynamic data structures.

2.4 Scaling the System

One of our main objectives was to design a system whose performance can be scaled up by adding additional low-cost workstations and using them to run additional components. Starting from the configuration in Figure 2, we could simply add additional downloaders and resolvers to improve performance. We estimate that a single manager would be fast enough for about 8 downloaders, which in turn would

[5] Available at `http://www.sleepycat.com`

[6] `http://www.chiark.greenend.org.uk/ ian/adns/`

[7] `http://users.footprints.net/ kaz/kazlib.html`

require maybe 2 or 3 DNS resolvers. Beyond this point, we would have to create a second crawl manager (and thus essentially a second crawling system), and the application would have to split its requests among the two managers. However, the first bottleneck in the system would arise before that, in our crawl application, which is currently able to parse and process up to 400 pages per second on a typical workstation. While this number could be improved somewhat by more aggressive optimizations, eventually it becomes necessary to partition the application onto several machines.

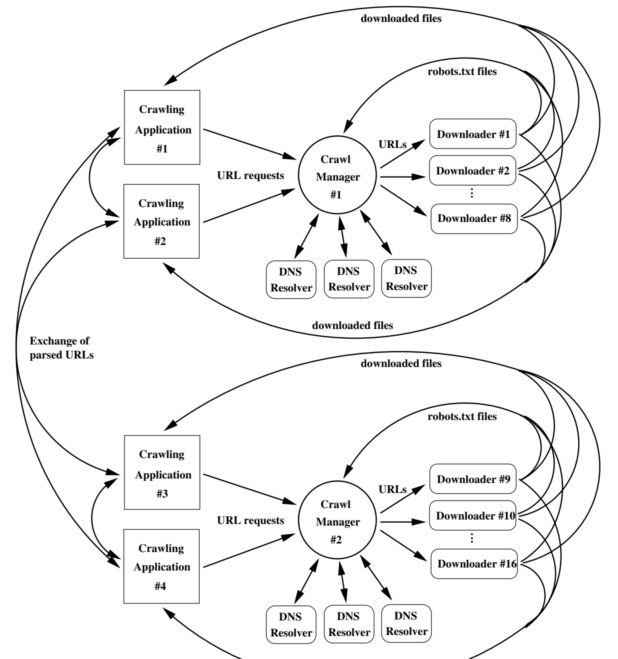


Figure 3: Large configuration

Figure 3 shows a possible scaled up version of our system that uses two crawl managers, with 8 downloaders and 3 DNS resolvers each, and with four application components. We point out that we have never been able to test the performance of such a configuration, which would involve about 20 machines and result in download rates of probably about 1500 pages per second, way beyond the capacity of our T3, or even a dedicated OC3 link.

Partitioning the breadth-first crawler into 4 components is quite simple, using a technique similar to that employed by the Internet Archive crawler [6]. We partition the space of all possible URLs into 4 subsets using a hash function, such that each application component is responsible for processing and requesting one subset. Recall that the manager will make sure that pages downloaded for different applications are stored in separate directories, as determined by the applications. If during parsing, a component encounters a hyperlink belonging to a different subset, then that URL is simply forwarded to the appropriate application component (as determined by the hash value). Each crawl manager could be used by two application components; to the manager, they would appear as completely unrelated applications. Undue load on any particular server could be avoided by either multiplying the 30 second interval by the number of crawl managers, or by making sure that each host is mapped to at most one subset; this second option would also avoid an increase in the cost of robot exclusion and DNS resolution.

Some other crawling strategies such as focused crawling may be harder to parallelize, but this is unavoidable and not particular to our design. We note that the only communication involving significant amounts of data is the transmission of the downloaded files. Thus, in principle our system could be used even in a wide-area distributed environment, assuming we can tolerate the fact that the downloaded files may end up in several remote locations (e.g., by collocating an application component at each downloader location).

Some more discussion on system configuration and resulting performance is given in Subsection 4.3.

3 Implementation Details and Algorithmic Techniques

We now describe some of the specific problems and performance bottlenecks that we encountered, and the data structures and techniques used to deal with them. We start out with the application and then follow the path of the URLs through the system. Recall that our breadth-first crawl application parses the newly downloaded documents for hyperlinks, then checks a data structure to see which of the URLs in the hyperlinks have already been encountered, and then sends the new URLs to the manager for downloading.

3.1 Application Parsing and Network Performance

Parsing is implemented using the Perl Compatible Regular Expression (PCRE) library[8]. Note that we only parse for hyperlinks, and not for indexing terms, which would significantly slow down the application. Parsing for indexing terms would have to be done by a separate application that scans the documents from the repository, and is outside the scope of this paper. Our current parsing implementation can process up to 400 pages per second at about 13 KB per page.

After tuning the parser, NFS and network performance considerations become more important on the application side. We usually have the downloader store the pages via NFS on a disk on the machine running the crawling application. Later, the application reads the files for parsing, and a storage manager copies them to a separate permanent repository, also via NFS. Thus, each data item has to enter and leave the machine running the application via the network. Our machines are connected by full-duplex switched Fast Ethernet. At 13 KB per page, maybe about 600 pages per

[8]Available at `http://www.pcre.org/`

second can enter and leave the application machine under realistic assumptions about network and NFS performance.

There are several possible solutions. We can scale the system by partitioning the application into several components as soon as the bottleneck arises, as described in Subsection 2.4. (We could also partition the downloaded files over several other machines, possibly at the downloaders, and have the application read via NFS, but this would not buy much.) Alternatively, an upgrade to Gigabit Ethernet and switch to using `rcp` should largely remove this bottleneck.

3.2 URL Handling

The hyperlinks parsed from the files, after normalization of relative links, are then checked against the "URL seen" structure that contains all URLs that have been downloaded or encountered as hyperlinks thus far[9]. A parsing speed of 300 pages per second results in more than 2000 URLs per second that need to be checked and possibly inserted. Each URL has an average length of more than 50 bytes, and thus a naive representation of the URLs would quickly grow beyond memory size.

Several solutions have been proposed for this problem. The crawler of the Internet Archive [6] uses a Bloom filter stored in memory; this results in a very compact representation, but also gives false positives, i.e., some pages are never downloaded since they collide with other pages in the Bloom filter. Lossless compression can reduce URL size to below 10 bytes [4, 24], though this is still too high for large crawls. In both cases main memory will eventually become a bottleneck, although partitioning the application will also partition the data structures over several machines. A more scalable solution uses a disk-resident structure, as for example done in Mercator [16]. Here, the challenge is to avoid a separate disk access for each lookup and insertion. This is done in Mercator by caching recently seen and frequently encountered URLs, resulting in a cache hit rate of almost 85%. Nonetheless, their system used several fast disks for an average crawl speed of 112 pages per second.

Our goal in the design was to completely avoid random disk accesses, or at least a linear growth of such accesses with the URLs. To achieve this, we perform the lookups and insertion in a bulk or offline operation. Thus, we take the following approach, based on well-known techniques: We initially keep the URLs in main memory in a Red-Black tree. When the structure grows beyond a certain size, we write the sorted list of URLs out to disk and switch to bulk mode. We now use the Red-Black tree in main memory to buffer newly encountered URLs, and periodically merge the memory-resident data into the disk-resident data using a simple scan-and-copy operation, during which all necessary lookups and insertions are performed. The merge is performed by spawning a separate thread, so that the application can continue parsing new files, and the next merge will only be started an hour after the previous one has completed. Thus, the system will gracefully adapt as the merge operations start taking longer while the structure grows. Efficiency is also optimized by storing URLs on disk in a simple compressed form, by removing common prefixes between subsequent URLs in the sorted order. Using this method, lookups and insertions were never a bottleneck.

Now recall that on average each page contains about 8 hyperlinks, and thus even after removing duplicates, the number of encountered but not yet downloaded pages will grow very quickly. After downloading 20 million pages, we have more than 100 million pages in the queue waiting to be downloaded. This has a number of consequences. First, any URL now parsed out of a page will only be downloaded in several days or weeks. Second, this justifies our decision to perform the lookup and insertion operations in a bulk fashion, since the URLs are not immediately needed by the crawling system. Moreover, if our goal is to only download 100 million pages, then we can switch off most of the application at this point, since the crawling system already has enough requests in the queue. Finally, it raises the question what exactly it means for the application to "keep up with the system". In our design of the breadth-first application, we have to parse all the files at the speed they are downloaded, so that they can afterwards be permanently transferred to storage. Alternatively, we could first store the pages, and retrieve and parse them later, as long as we can generate enough URLs to keep the crawling system busy.

3.3 Domain-Based Throttling

It is important not to put too much load on a single web server, by observing a time-out period between requests. It is also desirable to do domain-based throttling, to make sure that requests are balanced between different second- or higher-level domains. There are several motivations for this. Some domains may have a fairly slow network connection, but large number of web servers, and could be impacted by a crawler. Another problem we encountered was that larger organizations, such as universities, have intrusion detection systems that may raise an alarm if too many servers on campus are contacted in a short period of time, even if timeouts are observed between accesses to the same server. Finally, our crawler does timeouts between accesses based on hostname and not IP address, and does not detect if web servers are collocated on the same machine. In some cases, many hosts in a domain are collocated on one machine[10].

We decided to address domain-based throttling in the crawl

[9] We have not yet implemented any fingerprint technique, such as the one used in Mercator, to filter out downloaded pages with identical content but different URLs.

[10] We suspect that this is sometimes done to try to influence search engines with link-based ranking, since there were cases of such sites forming large cliques with very similar content or structure.

application, since this seemed to be the easiest way. We first note that fetching URLs in the order they were parsed out of the pages is a very bad idea, since there is a lot of second-level domain locality in the links. (Consider the case of the large cliques, which are sometimes attached to fairly cranky webmasters.) However, if we "scramble" the URLs into a random order, URLs from each domain will be spread out evenly. In fact, we can do this in a simple deterministic way with provable load balancing properties:

(1) Put the hostname of each URL into reverse order (e.g., `com.amazon.www`) before inserting the URL into the data structures of Subsection 3.2.

(2) After checking for duplicates, take the sorted list of new URLs and perform a k-way unshuffle permutation, say for $k = 1000$, before sending them to the manager.

A k-way unshuffle is easily implemented by scanning over the URLs in sorted order, dividing them among k files in a round-robin fashion, and concatenating these files. In fact, such unshuffle permutations have been used instead of random permutations in parallel computation [19] because of their balancing properties, which in our case guarantee that adjacent URLs in the sorted order are spread out far apart[11]. By reversing the hostname, pages in the same domain are spread out over the set of requests. We note that in our large crawl, we did not actually reverse the hostnames. We did not encounter any problems because of this, although this approach does not provide any provable bounds.

3.4 Crawl Manager Data Structures

The crawl manager maintains a number of data structures for scheduling the requests on the downloaders while observing robot exclusion and request interval policies. The main structures are shown in Figure 4. In particular, we have a FIFO request queue containing a list of the request files that have been sent by the manager. Each request file typically contains a few hundred or thousand URLs, for reasons of I/O efficiency, and is located on a disk accessible via NFS. Note that the files themselves are not immediately loaded by the managers, but stay on disk as long as possible.

Next, there are a number of FIFO host queues containing URLs, organized by hostname. These structures are implemented in Berkeley DB using a single B-tree, with hostname as a key, but the reader should think of them as separate queues for each host. Hosts themselves are organized in several host data structures; once a host has been selected for download, we take the first entry in the corresponding host queue and send it to a downloader. We have three different host structures: (i) a host dictionary containing an entry for each host currently in the manager, with a pointer to the corresponding host queue, (ii) a priority queue with pointers to

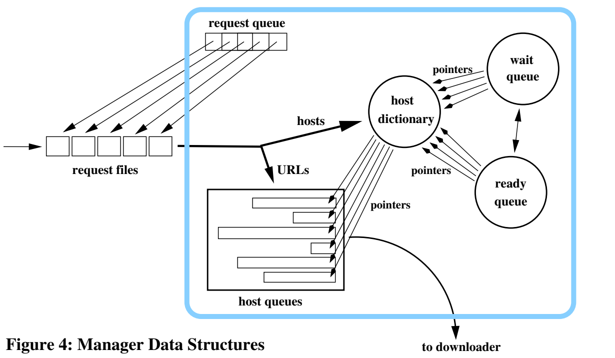


Figure 4: Manager Data Structures

those hosts that are ready for download (the *ready queue*), and (iii) a priority queue with pointers to those hosts that have recently been accessed and are now waiting for 30 seconds before they can be contacted again (the *waiting queue*). Each URL sent to the manager has an implicit request number corresponding to its ordering in the request stream. The goal of the manager is to preserve this ordering as much as possible, while observing the interval between requests and achieving high performance.

This is done as follows. Each host pointer in the ready queue has as its key value the request number of the first URL in the corresponding host queue. Thus, by extracting the host with minimum key value, we select the URL with the lowest request number among all URLs that are ready to be downloaded, and send it to the downloader. After the page has been downloaded, a pointer to the host is inserted into the waiting queue, with the key value equal to the time when the host can be accessed again. By checking on the minimum element in this priority queue, and extracting it if its waiting time has passed, we can transfer hosts back into the ready queue.

When a new host is encountered, we first create a host structure and put it into the host dictionary. Once DNS resolution and robot exclusion have been finished, we insert a pointer to the host into the ready queue. When all URLs in a queue have been downloaded, the host is deleted from the structures. However, certain information in the robots files is kept so that a host does not have to be recontacted when another URL comes in and the host structure is reinitialized. Finally, if a host is not responding, we put the host into the waiting queue for some time, and for an even longer time if it is down on the second try. Applications can also decide to give longer or shorter timeouts to certain hosts, e.g., to crawl a powerful server faster[12]. The priority queues and dictionary are implemented using STL.

[11] Note that a bit-reversal permutation has a similar effect.

[12] In [21] it is suggested to choose as timeout the time for the previous download to complete, multiplied by 10. This could be easily incorporated into our system.

3.5 Scheduling Policy and Manager Performance

We now discuss the scheduling policy and the performance of the manager data structures. As mentioned, the number of requested URLs that are waiting to be downloaded may be in the hundreds of millions, and the number of hosts may be several millions. If we immediately insert all the URLs into the Berkeley DB B-tree structure, it will quickly grow beyond main memory size and result in bad I/O behavior (we would expect a disk access for almost all URLs, especially after unshuffling the URL sets in the application).

Thus, we would like to delay inserting the URLs into the structures as long as possible. Recall that we have pointers to the URL files organized as a queue. We now follow a simple policy: Whenever there are less than x hosts in the ready queue, we read the next batch of URLs into the manager data structures. Our goal is to significantly decrease the size of the structures, or to at least get better I/O behavior. We choose some value $x > s \cdot t$, where s is the number of pages crawled per second, and t is the timeout interval for a host (30 seconds). Thus, as long as we have x hosts in this queue, we have enough hosts to keep the crawl running at the given speed. (We note that we do not count the hosts in the waiting queue, since it also contains hosts that are timed out for longer periods due to servers being down.) The total number of host structures and corresponding URL queues at any time is about $x + s \cdot t + n_d + n_t$, where x is the number of hosts in the ready queue, $s \cdot t$ is an estimate of the number of hosts currently waiting because they have recently been accessed, n_d is the number of hosts that are waiting because they were down, and n_t is the number of hosts in the dictionary that are in neither queue, which is small since this only happens during robot exclusion, DNS lookup, or the actual process of downloading a page. Ignoring n_d, which is hard to estimate, the number of host structures will usually be less than $2x$. For example, we typically used $x = 10000$, which for a speed of 120 pages per second resulted in at most 16000 hosts in the manager.

We now look at the consequences for I/O performance. As it turns out, even with this limited number of hosts, the B-tree for the URL queues in Berkeley DB will eventually grow beyond main memory size, due to a number of hosts with many URLs that are waiting to be downloaded and whose queues continue to grow. However, if we look at caching effects, we find that each queue has only two data pages that are really "active": one at the head, where URLs are removed, and one at the tail where new ones are inserted. (In fact, the head of one queue is likely to be on the same page as the tail of the next queue, and the structures of hosts waiting for longer times because they were down are rarely accessed.) Thus, if we make sure that there is enough main memory cache to hold most active data pages, we would expect that reasonable caching policies will work well. We used various sizes between 100 and 500 MB for Berkeley DB cache size, and always observed good behavior. As a result, the manager uses only moderate CPU, memory, and disk resources for a small number of downloaders.

We could now ask how this policy affects the order in which pages are sent to the downloaders. The answer is that the ordering is in fact the same as if we immediately insert all request URLs into the manager. Here, we assume that when the number of hosts in the ready queue drops below x, the manager will be able to increase this number again to at least x before the downloaders actually run out of work, which seems reasonable. (There is also a "host obliviousness" assumption essentially saying that changes in the exact timing of a host access should not result in widely different host response time.) Thus, we have the following result:

Any two requests for pages from the same host will be downloaded in the order they were issued by the application. Any two requests for pages from different hosts will be downloaded in the order they were issued, unless one host queue is "backed up" with requests due to the timeout rule or the host being down, or a delay occurred during robot exclusion, DNS resolution, or the actual process of downloading from a host. (The case of the delay should only matter if the requests were issued very close to each other.)

4 Experimental Results and Experiences

We now finally present some preliminary experimental results and experiences. A detailed analysis of performance bottlenecks and scaling behavior is beyond the scope of this paper, and would require a fast simulation testbed, since it would not be possible (or appropriate) to do such a study with our current Internet connection or on real web sites.

4.1 Results of a Large Crawl

We ran a crawl of over 120 million web pages on about 5 million hosts. The crawl was performed over a period of 18 days; however, the crawler was not continuously in operation during this time. There were three longer network outages, two of them due to outside attacks on the campus network. We had a number of crawler crashes in the beginning, since this was our first crawl of this size. The crawler was also offline for many hours while changes to the software were implemented. Finally, in the last 4 days, the crawler was running at very low speed to download URLs from a few hundred very large host queues that remained (the application was switched off much earlier after enough URLs had been discovered). After a crash, the crawler would recrawl a limited number of pages from the previous checkpoint. During operation, the speed of the crawler was limited by us to a certain rate, depending on time of day, to make sure that other users on campus were not (too much) inconvenienced. Table 4.1 shows approximate statistics for the crawl (some percentages were estimated from a subset of the logs).

HTTP requests	161,549,811
network errors	5,873,685
read_timeout_exceeded errors	2,288,084
robots.txt requests	16,933,942
successful non-robots requests	138,742,184
average size of page	13,354 bytes
total data size	1.85 TB

Table 4.1. Basic crawl statistics

total successfull non-robot	138,742,184	100.00%
200 (OK)	121,292,132	87.42%
404 (not found)	7,379,633	5.32%
302 (moved temporarily)	6,065,464	4.37%
301 (moved permanently)	2,885,665	2.08%
403 (forbidden)	488,132	0.35%
401 (unauthorized)	413,853	0.30%
500 (internal server error)	91,891	0.07%
other	125,414	0.09%

Table 4.2. HTTP errors

Network errors include a server that is down, does not exist, behaves incorrectly or is extremely slow (read_timeout_exceeded). Some robots files were downloaded many times, since they expire from the cache after 24 hours. We estimate that at least 120 million of the 138 million successful requests were to unique pages. Note that these numbers include HTTP errors, shown in Table 4.2:

4.2 Network Limits and Speed Control

As mentioned, we had to control the speed of our crawler so that impact on other campus users is minimized. To do this, we agreed with the network administrator on reasonable limits during different times of the day. We usually limited rates to about 80 pages per second ($1MB/s$) during peak times and up to 180 pages per second during the late night and early morning. These limits can be changed and displayed via a web-based Java interface. Our campus is connected to the Internet by a T3 link to AppliedTheory, with a Cisco 3620 as main campus router. We observed that this router gets clogged with requests as we approach the capacity of the T3. We briefly ran it at a rate of 300 pages per second, but doing this longer would probably bring down the router (again).

To achieve the speed control, the crawl manager varies the number of connections that each downloader opens simultaneously. Thus, if the measured download rate is below the target, the number is increased, and decreased otherwise. As it turns out, controlling speed automatically is quite challenging, since there are many factors that influence performance. Adding more and more connections when the campus network is busy or down is not a good idea, and decreasing the number of connections can in some cases increase speed, if the bottleneck was on the machines running the downloaders. We ended up with a not completely satisfactory solution that modifies the number of connections while also having an absolute upper bound on the number of connections for each speed setting.

In the next figure, we have the number of incoming bytes, outgoing bytes, and outgoing frames for our T3 campus connection, courtesy of AppliedTheory. This data includes all traffic going in and out of the `poly.edu` domain over the 24 hours of May 28, 2001. During this day, the crawler was run at high speed, and there was relatively little other traffic originating from campus. The crawler performed a checkpoint every 4 hours, during which crawling speed was close to zero for a few minutes. This is clearly shown in the graph for the number of incoming bytes, and the bottoms in this curve can be used to estimate the amount of bandwidth taken by the crawler versus the other campus traffic. The pattern does not exist in the outgoing bytes, since the crawler only sends out small requests, but is clearly visible in the number of outgoing frames, partly due to HTTP requests and the DNS system[13].

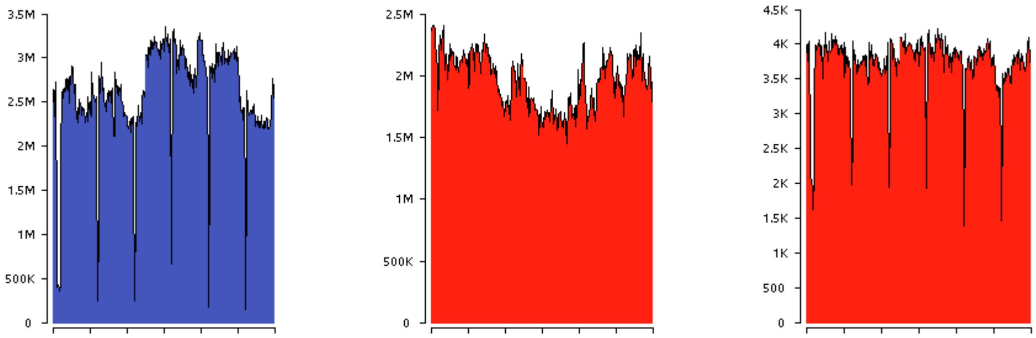
4.3 System Performance and Configuration

For the above crawl, we used 3 Sun Ultra10 workstations and a dual-processor Sun E250. We did not try to minimize the amount or cost of the hardware used, and the above configuration is powerful enough for much higher download rates on a fast enough network. We now try to give a rough estimate of what amount of hardware is really needed. This requires some understanding of the CPU, memory, and disk requirements of the different components, so that some of them can be collocated on the same machine. A downloader with a thousand connections will take up most of the CPU, but little memory. The manager takes only little CPU time (depending on how many downloaders are attached), and needs a reasonable amount (100 MB) of buffer space for Berkeley DB. The application needs both CPU and memory, and should thus not be collocated with a downloader.

As an example of a low-end configuration, we used two Sun Ultra 10 with 2×30 GB of EIDE disks each and 1 GB and 768 MB of memory, respectively. By running a downloader and the manager on one machine, and all other components on the other, we observed a sustained crawling speed of about 140 pages per second[14]; similar performance is also achievable on two comparable Linux workstations. (Attempts to increase this speed often result in sharply lower speed, due to components starving each other.)

[13] We also note that the number of outgoing bytes is unusually high in the figure, probably due to one of the mentioned break-ins that brought our campus network down several times.

[14] Actually, for the machine running the manager and downloader, about 256 MB of memory would suffice.



We note that this translates to a crawling speed of more than 12 million pages per day, enough for most academic research projects. For larger configurations, we believe that between 70 and 100 pages per second and per node should be achievable by distributing components appropriately, though this remains to be studied.

4.4 Other Experiences

As observed by Brin and Page [5], running a high-speed crawler "generates a fair amount of email and phone calls". This is still the case, and thus crawler supervision is very important. A small "host blacklist" data structure at the manager, which can be changed quickly to prevent the crawler from re-annoying already annoyed people, is highly useful. One issue we ran in repeatedly came from security software that raises an alarm when it believes someone is trying to break in by scanning ports on many machines, and several University networks alarmed our own network administrator about suspected attacks. (These tools are often unaware that they are dealing with HTTP requests.) Another company that tries to protect their customers' ports from scanning issued a stern alarm after only three accesses, which is problematic when the port protected in such a way formerly used to house a web server still linked to by over a hundred pages all over the web, according to AltaVista. Among other events were bugs in robot exclusion[15], helpful bug reports, and various curious questions. There are clearly numerous social and legal issues that arise.

5 Comparison with Other Systems

We now compare our architecture and performance with that of other systems that we know of. While all the major search engines, and a number of other navigation services and "copyright surveillance" companies, have their own high-performance crawling systems, most of the details of these systems are not public. The only detailed description of a high-performance crawler in the academic literature that we know is that of the *Mercator* crawler by Heydon and Najork [16], used by the AltaVista search engine. We give a detailed comparison with their system and with a recent successor system called *Atrax* [20] concurrently developed with our system. We also compare with certain aspects of other crawlers that we know about, though we are not familiar with many other aspects of these systems. For work on crawling strategies, see the references in Section 1.

Mercator was written completely in Java, which gives flexibility through pluggable components, but also posed a number of performance problems addressed in [16]. Mercator is a centralized crawler, designed for a fairly powerful server with several CPUs and fast disk. One major difference to our system is that we try to completely avoid random I/O, while Mercator uses caching to catch most of the random I/O, and a fast disk system to handle the still significant remaining accesses. An advantage of such a centralized system is that data can be directly parsed in memory and does not have to be written from disk as in our system. When scheduling URLs for download, Mercator obtains good I/O performance by hashing hostnames to a set of about 600 queues, instead of one queue for each host as we do. Mercator uses a thread for each queue that opens one connection using synchronous I/O, while we have a few fairly powerful downloaders that open hundreds of asynchronous connections.

A recent distributed version of Mercator called Atrax [20] essentially ties several Mercator systems together using the technique for partitioning the application described in Subsection 2.4, which has been earlier used in the Internet Archive crawler [6]. Atrax also uses a disk-efficient merge, similar to that described in Subsection 3.2. We are not yet familiar with many details of Atrax, which are still unpublished. However, one could say that Atrax employs a very similar approach for scaling, but uses a powerful centralized system (Mercator) as its basic unit of replication, while we replicate a small distributed system on a network of workstations, such as the one in Figure 2, to get a larger one as shown in Figure 3. (Another way of looking at the difference is to say that Atrax scales by replicating several vertical slices (Mercators), while our system layers several scalable horizontal slices (services) such as application, URL queuing in the manager, downloader.)

Some limited details on an early version of the Google crawler are given in [5]. The system also uses asynchronous

[15] It is surprisingly difficult to write a parser for robot files that works correctly on millions of servers, and a number of available libraries are buggy.

I/O in Python for its downloaders. One difference is that parsing for indexing terms is integrated with the crawling system, and thus download speed is limited by the indexing speed. The already mentioned Internet Archive crawler [6] uses a Bloom filter for identifying already seen pages, which allows the structure to be held in memory but also results in some pages being falsely omitted.

6 Conclusions and Future Work

We have described the architecture and implementation details of our crawling system, and presented some preliminary experiments. There are obviously many improvements to the system that can be made. A major open issue for future work is a detailed study of the scalability of the system and the behavior of its components. This could probably be best done by setting up a simulation testbed, consisting of several workstations, that simulates the web using either artificially generated pages or a stored partial snapshot of the web[16]. We are currently considering this, and are also looking at testbeds for other high-performance networked systems (e.g., large proxy caches).

Our main interest is in using the crawler in our research group to look at other challenges in web search technology, and several students are using the system and acquired data in different ways.

Acknowledgments: Thanks to our campus network administrator, Tom Schmidt, for his patience, and to Jeff Damens, Jeonghan Lim, and Jiwu Duan for great systems support. We also acknowledge equipment donations by Intel Corporation and Sun Microsystems.

References

[1] A. Arasu, J. Cho, H. Garcia-Molina, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technologies*, 1(1), June 2001.

[2] R. Baeza-Yates and B. Rebeiro-Neto. *Modern Information Retrieval*. Addision Wesley, 1999.

[3] Z. Bar-Yossef, A. Berg, S. Chien, J. Fakcharoenphol, and D. Weitz. Approximating aggregate queries about web pages via random walks. In *Proc. of 26th Int. Conf. on Very Large Data Bases*, September 2000.

[4] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The connectivity server: Fast access to linkage information on the web. In *7th Int. World Wide Web Conference*, May 1998.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the Seventh World-Wide Web Conference*, 1998.

[6] M. Burner. Crawling towards eternity: Building an archive of the world wide web. *Webtechniques*, 1997.

[7] S. Chakrabarti, B. Dom, R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins, D. Gibson, and J. Kleinberg. Mining the web's link structure. *IEEE Computer*, 32(8):60–67, 1999.

[8] S. Chakrabarti, M. van den Berg, and B. Dom. Distributed hypertext resource discovery through examples. In *Proc. of 25th Int. Conf. on Very Large Data Bases*, pages 375–386, September 1999.

[9] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *Proc. of the 8th Int. World Wide Web Conference (WWW8)*, May 1999.

[10] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proc. of 26th Int. Conf. on Very Large Data Bases*, pages 117–128, Sept. 2000.

[11] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 117–128, May 2000.

[12] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through url ordering. In *7th Int. World Wide Web Conference*, May 1998.

[13] M. Diligenti, F. Coetzee, S. Lawrence, C. Giles, and M. Gori. Focused crawling using context graphs. In *Proc. of 26th Int. Conf. on Very Large Data Bases*, September 2000.

[14] M. R. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork. Measuring index quality using random walks on the web. In *Proc. of the 8th Int. World Wide Web Conference (WWW8)*, pages 213–225, 1999.

[15] M. R. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork. On near-uniform URL sampling. In *Proc. of the 9th Int. World Wide Web Conference*, May 2000.

[16] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.

[17] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. WebBase : A repository of web pages. In *Proc. of the 9th Int. World Wide Web Conference*, May 2000.

[18] B. Kahle. Archiving the internet. *Scientific American*, March 1997.

[19] M. Kaufmann, J. F. Sibeyn, and T. Suel. Derandomizing algorithms for routing and sorting on meshes. In *Proc. of the 5th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 669–679, Arlington, VA, Jan. 1994.

[20] M. Najork. Atrax: A distributed web crawler. Presentation given at AT&T Research, March 20, 2001.

[21] M. Najork and J. Wiener. Breadth-first search crawling yields high-quality pages. In *10th Int. World Wide Web Conference*, 2001.

[22] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *Proc. of 27th Int. Conf. on Very Large Data Bases*, Sept. 2001. to appear.

[23] J. Rennie and A. McCallum. Using reinforcement learning to spider the web efficiently. In *Proc. of the Int. Conf. on Machine Learning (ICML)*, 1999.

[24] T. Suel and J. Yuan. Compressing the graph structure of the web. In *Proc. of the IEEE Data Compression Conference (DCC)*, March 2001. To appear.

[25] J. Talim, Z. Liu, P. Nain, and E. Coffman. Controlling robots of web search engines. In *SIGMETRICS Conference*, June 2001.

[26] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.

[16] Note that this is basically a scalable crawler trap.