WIRE: an Open Source Web Information Retrieval Environment

Carlos Castillo
Center for Web Research, Universidad de Chile
carlos.castillo@upf.edu

Ricardo Baeza-Yates
Center for Web Research, Universidad de Chile
ricardo.baeza@upf.edu

Abstract

In this paper, we describe the WIRE (Web Information Retrieval Environment) project and focus on some details of its crawler component. The WIRE crawler is a scalable, highly configurable, high performance, open-source Web crawler which we have used to study the characteristics of large Web collections.

1. Introduction

At the Center for Web Research (http://www.cwr.cl/) we are developing a software suite for research in Web Information Retrieval, which we have called **WIRE** (**W**eb **I**nformation **R**etrieval **E**nvironment). Our aim is to study the problems of Web search by creating an efficient search engine. Search engines play a key role on the Web, as searching currently generates more than 13% of the traffic to Web sites [1]. Furthermore, 40% of the users arriving to a Web site for the first time clicked a link from a search engine's results [14].

The WIRE software suite generated several sub-projects, including some of the modules depicted in Figure 1. So far, we have developed an efficient general-purpose Web crawler [6], a format for storing the Web collection, a tool for extracting statistics from the collection and generating reports and a search engine based on SWISH-E using PageRank with non-uniform normalization [3].

In some sense, our system is aimed at a specific segment: our objective was to use it to download and analyze collections having in the order of $10^6 - 10^7$ documents. This is bigger than most Web sites, but smaller than the complete Web, so we worked mostly with national domains (ccTLDs: country-code top level domains such as `.cl` or `.gr`). The main characteristics of the WIRE crawler are:

High-performance and scalability: It is implemented using about 25,000 lines of C/C++ code and designed to work with large volumes of documents and to handle up to a thousand HTTP requests simultaneously. The current implementation would require further work to scale to billions of documents (e.g.: process some data structures on disk instead of in main memory). Currently, the crawler is parallelizable, but unlike [8], it has a central point of control.

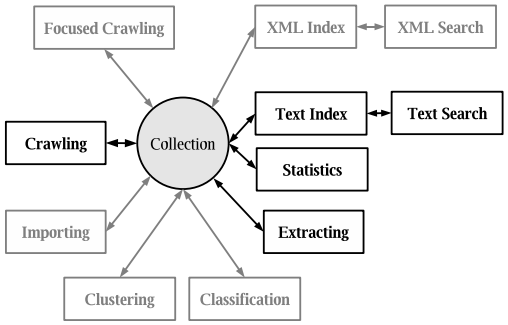

Figure 1. Some of the possible sub-projects of WIRE, highlighting the completed parts.

Configurable and open-source: Most of the parameters for crawling and indexing can be configured, including several scheduling policies. Also, all the programs and the code are freely available under the GPL license.

The details about commercial search engines are usually kept as business secrets, but there are a few examples of open-source Web crawlers, for instance Nutch http://lucene.apache.org/nutch/. Our system is designed to focus more on evaluating page quality, using different crawling strategies, and generating data for Web characterization studies. Due to space limitations, on this paper we describe only the crawler in some detail. Source code and documentation, are available at http://www.cwr.cl/projects/WIRE/.

The rest of this paper is organized as follows: section 2 details the main programs of the crawler and section 3 how statistics are obtained. The last section presents our conclusions.

2 Web crawler

In this section, we present the four main programs that are run in cycles during the crawler's execution: manager, harvester, gatherer and seeder, as shown in Figure 2.

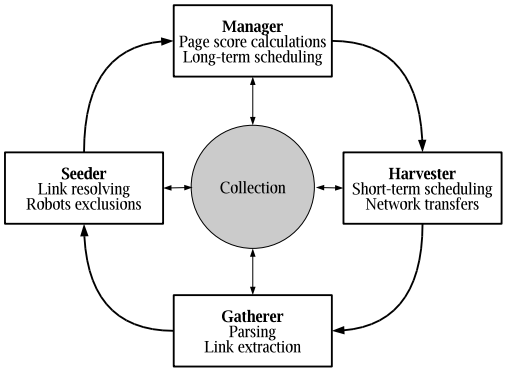

Figure 2. Modules of the crawler.

2.1 Manager: long-term scheduling

The "manager" program generates the list of K URLs to be downloaded in this cycle (we used $K = 100,000$ pages by default). The procedure for generating this list is based on maximizing the "profit" of downloading a page [7].

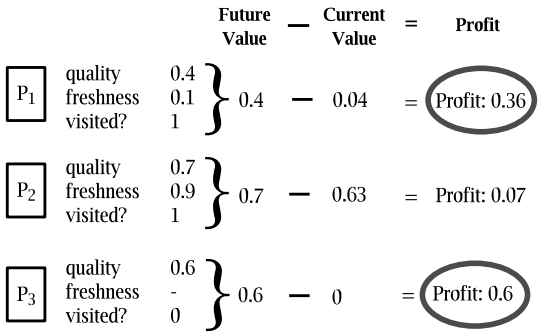

Figure 3. Operation of the manager program.

The current value of a page depends on an estimation of its intrinsic quality, and an estimation of the probability that it has changed since it was crawled.

The process for selecting the pages to be crawled next includes (1) filtering out pages that were downloaded too recently, (2) estimating the quality of Web pages, (3) estimating the freshness of Web pages and (4) calculating the profit for downloading each page. This balances the process of downloading new pages and updating the already-downloaded ones. For example, in Figure 3, the behavior of the manager for $K = 2$ is depicted. In the figure, it should select pages P_1 and P_3 for this cycle as they give the higher profit.

2.2 Harvester: short-term scheduling

The "harvester" program receives a list of K URLs and attempts to download them from the Web. The politeness policy chosen is to never open more than one simultaneous connection to a Website, and to wait a configurable amount of seconds between accesses (default 15). For the larger Websites, over a certain quantity of pages (default 100), the waiting time is reduced (to a default of 5 seconds).

As shown in Figure 4, the harvester creates a queue for each Web site and opens one connection to each active Web site (sites 2, 4, and 6). Some Web sites are "idle", because they have transfered pages too recently (sites 1, 5, and 7) or because they have exhausted all of their pages for this batch (3). This is implemented using a priority queue in which Web sites are inserted according to a time-stamp for their next visit.

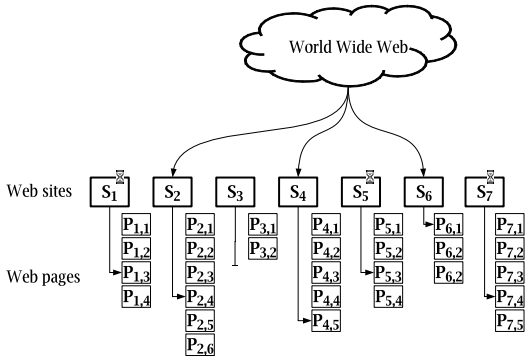

Figure 4. Operation of the harvester program.

Our first implementation used Linux threads and did blocking I/O on each thread. It worked well, but was not able to go over 500 threads even in PCs with processors of 1 GHz and 1GB of RAM. It seems that entire thread system was designed for only a few threads at the same time, not for higher degrees of parallelization. Our current implementation uses a single thread with non-blocking I/O over an array of sockets. The `poll()` system call is used to check for activity in the sockets. This is much harder to implement than the multi-threaded version, as in practical terms it involves programming context switches explicitly, but the performance is much better, allowing us to download from over 1000 Web sites at the same time with a very lightweight process.

2.3 Gatherer: parsing of pages

The "gatherer" program receives the raw Web pages downloaded by the harvester and parses them. In the current implementation, only HTML and plain text pages are accepted by the harvester.

The parsing of HTML pages is done using an events-oriented parser. An events-oriented parser (such as SAX [12] for XML) does not build an structured representation of the documents: it just generates function calls whenever certain conditions are met. We found that a substantial amount of pages were not well-formed (e.g.: tags were not balanced), so the parser must be very tolerant to malformed markup.

The contents of Web pages are stored in variable-sized records indexed by document-id. Insertions and deletions are handled using a free-space list with first-fit allocation. This data structure also implements duplicate detection: whenever a new document is stored, a hash function of its contents is calculated. If there is another document with the same hash function and length, the contents of the documents are compared. If they are equal, the document-id of the original document is returned, and the new document is marked as a duplicate.

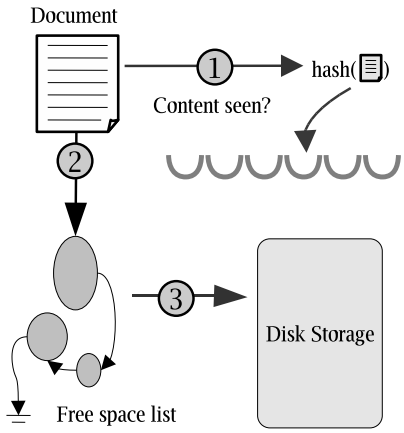

Figure 5. Storing the contents of a document.

The process for storing a document, given its contents and document-id, is depicted in Figure 5. For storing a document, the crawler has to check first if the document is a duplicate, then search for a place in the free-space list, and then write the document to disk. This module requires support to create large files, as for large collections the disk storage grows over 2GB, and the offset cannot be provided in a variable of type "`long`". In Linux, the LFS standard [10] provides offsets of type "`long long`" that are used for disk I/O operations. The usage of continuous, large files for millions of pages, instead of small files, can save a lot of disk seeks, as noted also by Patterson [16].

2.4 Seeder: URL resolver

The "seeder" program receives a list of URLs found by the gatherer, and adds some of them to the collection, according to a criteria given in the configuration file. This criteria includes patterns for accepting, rejecting, and transforming URLs.

Patterns for **accepting** URLs include domain name and file name patterns. The domain name patterns are given as suffixes (e.g.: `.cl`, `.uchile.cl`, etc.) and the file name patterns are given as file extensions. Patterns for **rejecting** URLs include substrings that appear on the parameters of known Web applications (e.g. `login`, `logout`, `register`, etc.) that lead to URLs which are not relevant for a search engine. Finally, to avoid duplicates from session ids, patterns for **transforming** the URLs are used to remove known session-id variables such as `PHPSESSID` from the URLs.

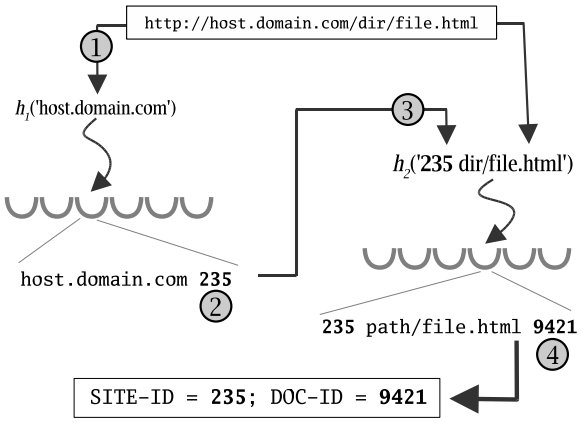

Figure 6. For checking a URL: (1) the host name is searched in the hash table of Web site names. The resulting site-id (2) is concatenated with the path and filename (3) to obtain a doc-id (4).

The structure that holds the URLs is highly optimized for the most common operations during the crawling process: given the name of a Web site, obtain a site-id, given the site-id of a Web site and a local link, obtain a document-id, and given a full URL, obtain both its site-id and document-id. The process for converting a full URL is shown in Figure 6.

This process is optimized to exploit the locality on Web links, as most of the links found in a page point to other pages co-located in the same Web site. For this, the implementation uses two hash tables: the first for converting

Web site names into site-ids, and the second for converting "site-id + path name" to a doc-id.

3 Obtaining statistics

To run the crawler on a large collection, the user must specify the site suffix(es) that will be crawled (e.g.: `.kr` or `.upf.edu`), and has to provide a starting list of "seed" URLs. Also, the crawling limits have to be provided, including the maximum number of pages per site (the default is 25,000) and the maximum exploration depth (default is 5 levels for dynamic pages and 15 for static pages).

There are several configurable parameters, including the amount of time the crawler waits between accesses to a Web site –that can be fine-tuned by distinguishing between "large" and "small" sites– the number of simultaneous downloads, the timeout for downloading pages, among many others. On a standard PC with a 1 GHz Intel 4 processor and 1 GB of RAM, using standard IDE disks, we usually download and parse about 2 million pages per day.

WIRE stores as much metadata as possible about Web pages and Web sites during the crawl, and includes several tools for extracting this data and for obtaining statistics. The analysis includes running link analysis algorithms such as Pagerank [15] and HITS [11], aggregating this information by documents and sites, and generating histograms for almost every property that is stored by the system. It also includes a module for detecting the language of a document based on a dictionary of stopwords in several languages that is included with WIRE.

The process for generating reports includes the analysis of the data, its extraction, the generation of `gnuplot` scripts for plotting, and the compilation of automated reports using LaTeX. The generated reports include: distribution of language, histograms of in- and out-degree, link scores, page depth, HTTP response codes, age (including per-site average, minimum and maximum), summations of link scores per site, histogram of pages per site and bytes per site, an analysis by components in the Web structure [5], the distribution of links to multimedia files, and of links to domains that are outside the delimited working set for the crawler.

4 Conclusions

So far, we have used WIRE to study large Web collections including the national domains of Brazil [13], Chile [2], Greece [9] and South Korea [4]. We are currently developing a module for supporting multiple text encodings including Unicode.

While downloading a few thousands pages from a bunch of Web sites is relatively easy, building a Web crawler that has to deal with millions of pages and also with misconfigured Web servers and bad HTML coding requires solving a lot of technical problems. The source code and the documentation of WIRE, including step-by-step instructions for running a Web crawl and analysing the results, are available at `http://www.cwr.cl/projects/WIRE/doc/`.

References

[1] Search Engine Referrals Nearly Double Worldwide. http://websidestory.com/pressroom/pressreleases.html?id=181, 2003.

[2] R. Baeza-Yates and C. Castillo. Características de la Web Chilena 2004. Technical report, Center for Web Research, University of Chile, 2005.

[3] R. Baeza-Yates and E. Davis. Web page ranking using link attributes. In *Alternate track papers & posters of the 13th international conference on World Wide Web*, pages 328–329, New York, NY, USA, 2004. ACM Press.

[4] R. Baeza-Yates and F. Lalanne. Characteristics of the Korean Web. Technical report, Korea–Chile IT Cooperation Center ITCC, 2004.

[5] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the Web: Experiments and models. In *Proceedings of the Ninth Conference on World Wide Web*, pages 309–320, Amsterdam, Netherlands, May 2000. ACM Press.

[6] C. Castillo. *Effective Web Crawling*. PhD thesis, University of Chile, 2004.

[7] C. Castillo and R. Baeza-Yates. A new crawling model. In *Poster proceedings of the eleventh conference on World Wide Web*, Honolulu, Hawaii, USA, 2002.

[8] J. Cho and H. Garcia-Molina. Parallel crawlers. In *Proceedings of the eleventh international conference on World Wide Web*, pages 124–135, Honolulu, Hawaii, USA, 2002. ACM Press.

[9] E. Efthimiadis and C. Castillo. Charting the Greek Web. In *Proceedings of the Conference of the American Society for Information Science and Technology (ASIST)*, Providence, Rhode Island, USA, November 2004. American Society for Information Science and Technology.

[10] A. Jaeger. Large File Support in Linux. http://www.suse.de/aj/linux_lfs.html, 2004.

[11] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[12] D. Megginson. Simple API for XML (SAX 2.0). http://sax.sourceforge.net/, 2004.

[13] M. Modesto, á. Pereira, N. Ziviani, C. Castillo, and R. Baeza-Yates. Un novo retrato da Web Brasileira. In *Proceedings of SEMISH*, São Leopoldo, Brazil, 2005.

[14] J. Nielsen. Statistics for Traffi c Referred by Search Engines and Navigation Directories to Useit. http://www.useit.com/-about/searchreferrals.html, 2003.

[15] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the Web. Technical report, Stanford Digital Library Technologies Project, 1998.

[16] A. Patterson. Why writing your own search engine is hard. *ACM Queue*, April 2004.