International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 03 Issue 09
May 2016

Offline/Online Semantic Web Crawler

Nikita Suryavanshi; Deeksha Singh & Sunakashi
Galgotias College Of Engineering And Technology, Knowledge Park II, Greater Noida

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.

Search engine come to our rescue in such cases .with a search engine ,all the students has to do is type in the "keyword" relating to the information that he needs .The search engine would then return a set of results that match best with the keywords entered.

A Web search engine can therefore be defined as a software program at takes input from the user, searches its database and returns a set of results .It is important to note here that the search engine does not search the internet: rather it searches its database ,which is populated with data from the internet by its crawler .Therefore ,we chose to develop web search engine and the ranking method to arrange the pages found by search engine relevantly. So that the user who entered the query can find the most relevant page first (page which consist of relevant information required by user) .Our project has a feature called "Page Rank & Hits" that allows user to receive most relevant result in response to a query .For instance if user enters a keyword "student" as his query the pages consisting relevant information will be searched and then ranked according to their hit ratio and the page rank.

Keyword- ASP.NET; Visual Studio 2008; Visual Management Studio; My SQL

I. INTRODUCTION

Web *crawlers* are programs that exploit the graph structure of the Web to move from page to page. In their infancy such programs were also called wanderers, robots, spiders, fish, and worms, words that are quite evocative of Web imagery. It may be observed that the noun "crawler" is not indicative of the speed of these programs, as they can be considerably fast. In our own experience, we have been able to crawl up to tens of thousands of pages within a few minutes while consuming a small fraction of the available bandwidth.4

From the beginning, a key motivation for designing Web crawlers has been to retrieve Web pages and add them or their representations to a local repository. Such a repository may then serve particular application needs such as those of a Web search engine. In its simplest form a crawler starts from a *seed* page and then uses the external links within it to attend to other pages. The process repeats with the new pages offering more external links to follow, until a sufficient number of pages are identified or some higher-level objective is reached. Behind this

simple description lies a host of issues related to network connections, spider traps, canonicalizing URLs, parsing HTML pages, and the ethics of dealing with remote Web servers. In fact, a current generation Web crawler can be one of the most sophisticated yet fragile parts of the application in which it is embedded. Were the Web a static collection of pages we would have little long-term use for crawling. Once all the pages had been fetched to a repository (like a search engine's database), there would be no further need for crawling. However, the Web is a dynamic entity with subspaces evolving at differing and often rapid rates. Hence there is a continual need for crawlers to help applications stay current as new pages are added and old ones are deleted, moved or modified.

General-purpose search engines serving as entry points to Web pages strive for coverage that is as broad as possible. They use Web crawlers to maintain their index databases , amortizing the cost of crawling and indexing over the millions of queries received by them. These crawlers are blind and exhaustive in their approach, with comprehensiveness as their major goal. In contrast, crawlers can be selective about the pages they fetch and are then referred to as *preferential* or heuristic-based crawlers. These may be used for building focused repositories, automating resource discovery, and facilitating software agents. There is a vast literature on preferential crawling applications including [15, 9, 31, 20, 26, 3]. Preferential crawlers built to retrieve pages within a certain topic are called *topical* or *focused* crawlers. Synergism between search engines and topical crawlers is certainly possible, with the latter taking on the specialized responsibility of identifying subspaces relevant to particular communities of users. Techniques for preferential crawling that focus on improving the "freshness" of a search engine have also been suggested.

II. Building a Crawling Infrastructure

Figure 1 shows the flow of a basic sequential crawler (in Sect. 2.6 we consider multithreaded crawlers). The crawler maintains a list of unvisited URLs called the *frontier*.

The list is initialized with seed URLs, which may be provided by a user or another program. Each *crawling loop* involves picking the next URL to crawl from the frontier, fetching the page corresponding to the URL through HTTP, parsing the retrieved page to extract the URLs and application-specific information, and finally adding the unvisited URLs to the frontier. Before the URLs are added to the frontier they may be assigned a score that represents the estimated benefit of visiting the page corresponding to the URL. The crawling process may be terminated when a certain number of pages have been crawled. If the crawler is ready to crawl another page and the frontier is empty, the situation signals a deadend for the crawler. The crawler has no new page to fetch, and hence it stops. Crawling can be viewed as a graph search problem. The Web is seen as a large graph with pages at its nodes and hyperlinks as its edges. A crawler starts at a few of the nodes (seeds) and then follows the edges to reach other nodes. The process of fetching a page and extracting the links within it is analogous to expanding a node in graph search. A topical crawler tries to follow edges that are expected to lead to portions of the graph that are relevant to a topic

- **Frontier**

The frontier is the to-do list of a crawler that contains the URLs of unvisited pages. In graph search terminology the frontier is an *open list* of unexpanded (unvisited) nodes. Although it may be necessary to store the frontier on disk for large - scale crawlers, we will represent the frontier as an in-memory data structure for simplicity. Based on the available memory, one can decide the maximum size of the frontier. Because of the large amount of memory available on PCs today, a frontier size of a 100,000 URLs or more is not

exceptional. Given a maximum frontier size we need a mechanism to decide which URLs to ignore when this limit is reached. Note that the frontier canfill rather quickly as pages are crawled. One can expect around 60,000 URLs in the frontier with a crawl of 10,000 pages, assuming an average of about 7 links per page

- **History and Page Repository**

The crawl history is a time-stamped list of URLs that were fetched by the crawler. In effect, it shows the path of the crawler through the Web, starting from the seed pages. A URL entry is made into the history only after fetching the corresponding page. This history may be used for post-crawl analysis and evaluations. For example, we can associate a value with each page on the crawl path and identify significant events (such as the discovery of an excellent resource). While history may be stored occasionally to the disk, it is also maintained as an in-memory data structure. This provides for a fast lookup to check whether a page has been crawled or not. This check is important to avoid revisiting pages and also to avoid adding the URLs of crawled pages to the limited size frontier. For the same reasons it is important to canonicalize the URLs (Sect. 2.4) before adding them to the history.

Once a page is fetched, it may be stored/indexed for the master application (such as a search engine). In its simplest form a page repository may store the crawled pages as separate files. In that case, each page must map to a unique file name. One

way to do this is to map each page's URL to a compact string using some form of hashing function with low probability of collisions (for uniqueness of file names). The resulting hash value is used as the file name.

- **Fetching**

In order to fetch a Web page, we need an HTTP client that sends an HTTP request for a page and reads the response. The client needs to have timeouts to make sure that an unnecessary amount of time is not spent on slow servers or in reading large pages. In fact, we may typically restrict the client to download only the first 10–20KB of the page. The client needs to parse the response headers for status codes and redirections. We may also like to parse and store the last-modified header to determine the age of the document. Error checking and exception handling are important during the page-fetching process since we need to deal with millions of remote servers using the same code. In addition, it may be beneficial to collect statistics on timeouts and status codes for identifying problems or automatically changing timeout values. Modern programming languages such as Java and Perl provide very simple and often multiple programmatic interfaces for fetching pages from the Web. However, one must be careful in using high-level interfaces where it may be harder to find lower-level problems. For example, with Java one may want to use the java.net. Socket class to send HTTP requests instead of using the more ready-made java.net. HttpURLConnection class.

- **Parsing**

Once a page has been fetched, we need to parse its content to extract information that will feed and possibly guide the future path of the crawler. Parsing may imply simple hyperlink/URL extraction or it may involve the more complex process of tidying up the HTML content in order to analyze the HTML tag tree . Parsing might also involve steps to convert the extracted URL to a canonical form, remove stop words from the page's content, and stem the remaining words. These components of parsing are described next

- **Multithreaded Crawlers**

The multithreaded crawler model needs to deal with an empty frontier just like a sequential crawler. However, the issue is less simple now. If a thread finds the frontier empty, it does not automatically mean that the crawler as a whole has reached a dead end. It is possible that other threads are fetching pages and may add new URLs in the

near future. One way to deal with the situation is by sending a thread to a sleep state when it sees an empty frontier. When the thread wakes up, it checks again for URLs. A global monitor keeps track of the number of threads currently sleeping. Only when all the threads are in the sleep state does the crawling process stop. More optimizations can be performed on the multithreaded model described here, as for instance to decrease contentions between the threads and to streamline network access

III . Evaluation of Crawlers

In a general sense, a crawler (especially a topical crawler) may be evaluated on its ability to retrieve "good" pages.However, a major hurdle is the problem of recognizing these good pages. In an operational environment real users may judge the relevance of pages as these are crawled, allowing us to determine if the crawl was successful or not. Unfortunately, meaningful experiments involving real users for assessing Web crawls are extremely problematic. For instance, the very scale of the Web suggests that in order to obtain a reasonable notion of crawl effectiveness one must conduct a large number of crawls, i.e., involve a large number of users.

Second, crawls against the live Web pose serious time constraints. Therefore crawls other than short-lived ones will seem overly burdensome to the user. We may choose to avoid these time loads by showing the user the results of the full crawl but this again limits the extent of the crawl.

III. APPLICATION

Crawling in general and topical crawling in particular is being applied for various other applications, many of which do not appear as technical papers. For example, business intelligence has much to gain from topical crawling. A large number of companies have Web sites where they often describe their current objectives, future plans, and product lines. In some areas of business, there are a large number of start-up companies that have rapidly changing Web sites. All these factors make it important for various business entities to use sources other than the general-purpose search engines to keep track of relevant and publicly available information about their potential competitors or collaborators. Crawlers have also been used for biomedical applications like finding relevant literature on a gene. On a different note, there are some controversial applications of crawlers such as extracting e-mail addresses from Web sites for spamming.

IV. SUMMARY ANALYSIS

Given a particular measure of page importance we can summarize the performance of the crawler with metrics that are analogous to the information retrieval (IR) measures of *precision* and *recall*. Precision is the fraction of retrieved (crawled) pages that are relevant, while recall is the fraction of relevant pages that are retrieved (crawled). In a usual IR task the notion of a relevant set for recall is restricted to a given collection or database. Considering the Web to be one large collection, the relevant set is generally unknown for most Web IR tasks. Hence, explicit recall is hard to measure. Many authors provide precision-like measures that are easier to compute in order to evaluate the crawlers. We will discuss a few such precision-like measures:

1. *Acquisition rate*: In cases where we have Boolean relevance scores we could measure the explicit rate at which "good" pages are found. Therefore, if 50relevant pages are found in the first 500 pages crawled , then we have an acquisition

rate or *harvest rate* [1] of 10% at 500 pages.

2. *Average relevance*: If the relevance scores are continuous they can be averaged over the crawled pages. This is a more general form of harvest rate. The scores may be provided through simple cosine similarity or a trained classifier. Such averages (may be computed over the progress of the crawl (first100 pages, first 200 pages, and so on). Sometimes running averages are calculated over a window of a few pages (e.g., the last 50 pages from a current crawl point) .Since measures analogous to recall are hard to compute for the

Web, authors resort to indirect indicators for estimating recall. Some such indicators are:

1. *Target recall*: A set of known relevant URLs is split into two disjoint sets–*targets* and *seeds*. The crawler is started from the seeds pages and the recall of the targets is measured. The target recall is computed as *target recall* =

$$\frac{|Pt \cap Pc|}{|Pt|}$$

where *Pt* is the set of target pages, and *Pc* is the set of crawled pages *Robustness*: The seed URLs are split into two disjoint sets *Sa* and *Sb*. Each set is used to initialize an instance of the same crawler. The overlap in the pages crawled starting from the two disjoint sets is measured. A large overlap is interpreted as *robustness* of the crawler in covering relevant portions of the Web [9, 6]. There are other metrics that measure the crawler performance in a manner that combines both precision and recall. For example, *search length* [21] measures the number of pages crawled before a certain percentage of the relevant pages are retrieved.

V. Research Scope

As, the defined concepts for web crawling and improving its performance by the various crawling algorithms have been explained here. It has not end of the work for improving performance of crawling. There are many more techniques and algorithms may be considered for crawler to improve its performance. We can also improve its performance to modify the sitemap of any website, i.e. in sitemap protocol all URL has a static priority and we can change it by dynamic priority and this priority is calculated through user interest i.e. number of hits has high priority.

VI. Conclusion

Crawling: The websites submitted to the Crawler were crawled without any issues. The number of WebPages and the rates, at which they crawled, depends on the speed of the internet.

Searching: All the search results in response to a query are successfully retrieved. The time taken for the retrieval of results is a function of the size of the database.

Ranking: On the search query, an effective ranking algorithm is then applied so that the result should appear in relevant order.

So, we have achieved our aim by developing a search tool that gives the most relevant output in response to a query. The project developed by us is portable, cost-effective and efficient. It also has a user friendly interface.

The paper surveys several crawling methods or algorithms that are used for downloading the web pages from the World Wide Web. We believe that all of the algorithms discuss in this paper are well effective and high performance for web search, reduce the network traffic and crawling costs, but overall advantages and disadvantage favor more for By using HTTP Get Request and also Dynamic Web Page and download updated web pages By the using of filter is produce relevant results.

VI. References

[1] Internet theory available at, **http://en.wikipedia.org/wiki/Internet**

[2] Search Engine theory available at, **http://en.wikipedia.org/wiki/searchengine**

[3] Web Crawler theory available at ,**http://en.wikipedia.org/wiki/webcrawler**

[4] Benefits of google search engine available at, **http://www.google.co.uk/ technology /whyuse.html**

[5] Benefits of having a high page rank available at, **http://www.blong.info/ benefits-high-page-rank-pr.php**

[6] Page Rank theory available at, **http://en.wikipedia.org/wiki/PageRank**

p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 03 Issue 09
May 2016

[7] K.K.Aggarwal & Yogesh Singh, "Software Engineering", New Age International Publishers, Seventh Edition, 2009

[8] Bob Hughes & Mike Cotterell, "Software Project Management", Published by Tata McGraw-Hill Publishing Company Limited, Fourth Edition, 2009

[9] Donis Marshall, "Programming Microsoft Visual C# 2008: The Language", WP Publishers & Distributors(P) Limited, Third Edition 2008.